Universitat Politècnica de Catalunya

Master Thesis

Inferring program structure from execution traces

Author:
Juan Francisco Martínez Vera
juan.martinez@bsc.es

Supervisor:
Jesús Labarta Mancho
jesus.labarta@bsc.es

A thesis submitted in fulfillment of the requirements for the degree of Master in Research and Innovation

in the

Performance tools team
Barcelona Supercomputing Center

April 16, 2018

"Compró Geometría de Descartes y la leyó por si mismo. Cuando había leído dos o tres páginas, se sintió incapaz de seguir adelante. Empezó de nuevo y avanzó tres o cuatro páginas más, hasta llegar a otro punto dificil. Volvió a empezar y avanzó un poco más. Y continuó así hasta hacerse dueño de todo su significado."

Richard S. Westfall – Isaac Newton: Una vida

Abstract

Performance tools have been demonstrated to be valuable on detection of bottlenecks for a long time but since it demands high skilled analyst, developers are not used to use them but delegate this work to the actual specialist. What it means is that analyzers have to work with codes they are not familiar with. Having the structure of the program will lead to better understand the application by allowing to analyst to have a mental schema about what the program does. This improve in understandability also will leads to better and faster analysis while also drives to better reports to the developer. Identifying this need, in this thesis we propose a new methodology for application structure detection. Even if this problem has been previously solved by different researchers, they used to use sequential pattern mining techniques what are difficultly scalable what is a big drawback if we pay attention to the trend of the HPC systems to grow in complexity. We propose a disruptive design being the key point of the proposal the fact that we understand this problem as a classification problem instead of a sequential pattern mining problem. The main reason to do that is because we are exclusively focused on HPC applications so we decided to exploit their idiosyncrasy that have leads to a sort of ad-hoc methodology for HPC application that have allowed to decrease the complexity of the tool.

A prototype of the proposed methodology have been developed and tested demonstrating that is able to provide correct results for current real world applications with good scalability.

Contents

List of Figures

Chapter 1

Introduction

On the context of this thesis the motivations that encourage for this work and the objectives.

1.1 Context

From several decades to nowadays science have been evolving dramatically in a wide variety of fields, one of the main reasons is because now we have the technology to provide the enough computational power to face problems which were typically impossible to solve. The vanguard of this technological revolution is represented by huge computers with high resources availability such as processors, memory, disk or network. In Computer Science the discipline in charge to drive research in this field is the High Performance Computing, i.e. HPC. HPC has been increasing in importance and nowadays can be said it is the third support of science with theory and mathematics. The science that can be done thanks to this big machines goes from earthquake predictions to the analysis of the DNA of a carcinogenic cell, from weather forecast to material physics simulation.

A typical configuration for high performance machines are clusters. It means a team of individual processors or multiprocessors (and lately more specific hardware like GPUs) working together, interconnected by a super-fast network. The fundamental idea behind these big machines is getting speedup by mean of partitioning the problem and parallelize the execution. So all processors (or a subset) in the cluster will be dealing with different parts of the same problem and communicating between them in order to end up with a solution. For example imagine we have a weather forecast software, in order to speedup the forecast we can partition the surface of the earth and hold every portion to one individual processor. The communications will be needed because the weather will also depend on surroundings, so every partition will need also information of other partitions results.

Resources are limited and expensive so we have to use them as efficiently as its possible. Improving the performance and the efficiency is not just a matter that affects to one layer of the Transformation Hierarchy (proposed by [Patt, 2017], figure 1.2) but can be applied to every one of them. For example the tremendous evolution of the last fourty years were improvements done mostly on circuits layer what have been following the Moore's law [Moore, 1965]. The

Figure 1.1: Gordon Moore's prediction done in 1965

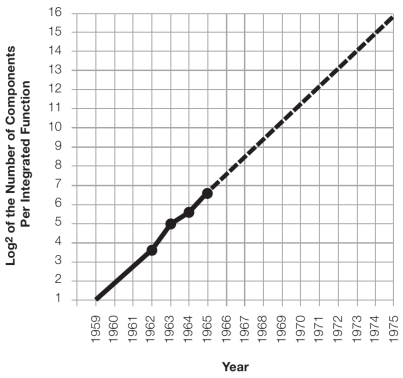
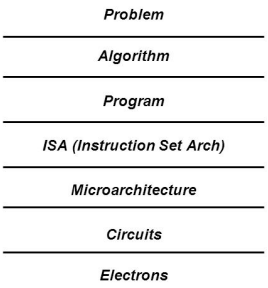


Figure 1.2: Levels of transformation



manufacturers have been reducing the size of transistors by a ratio of 2 every 18 months as were predicted in figure 1.1. Performance improvements were also at microarchitecture level with disruptive designs that allows ILP like HPS[1] [Patt et al., 1985], speculative execution with prefetchers, branch prediction or even memory access value speculation, VLIW[2] or TLP[3] with multi-threading. Improvements on memory hierarchy like cache associativity, non-blocking cache or trace cache. The last revolution affected microarchitecture, architecture and program layers was the introduction of multicores (early 2000s). It is basically a revolution in commodity hardware because HPC have been used to use shared memory paradigm for a while like for example with Convex (1991s). Anyway it is important even in HPC since the trend have been to move from ad-hoc systems to commodity hardware because the goodnesses of mass production. On last years researchers have been concerned about the power consumption because the trend is to have bigger machines so the need of be more energy efficient is a big deal for the next-generation exascale systems. Heterogeneity of systems seems to be the trend, i.e. near the typical general purpose chips install new specific purpose hardware like GPUs or ASICs like TPUs [Jouppi et al., 2017] evolving to an even more complex systems. This innovation affects to almost all layers of the stack from Program to below.

[1] High Performance Substrace, what is indeed out-of-order execution with in-order retirement.
[2] Very Long Instruction Word
[3] Thread Level Parallelism

Program layer[4] is the interface between pure algorithm and the machine architecture. It is the actual implementation of the algorithm, i.e. where programmers transforms algorithm to semantic code that can be transformed lately into executable binary by compilers. Having an efficient algorithm mathematically speaking is not enforcing having an efficient program because at algorithm level we do not care about the machine. Pushing on with the business to be always more faster and efficient, the tools that aims to this end in program layer are the performance analysis tools that allows to detect the bottlenecks presents in the application that prevents from better behaviour.

Once the big picture have been briefly introduced now we can zoom in and go a bit further into details. This thesis belongs to the application performance analysis field of research contributing to ease the analysis and the report. On the next section a more wide description on performance analysis tools is presented.

1.1.1 Performance analysis tools

There are two main trends on analysis tools:

i) Profiler tools generates performance summaries that provides coarse-grain information, with low-overhead. It is suitable for these analysis where you do not care about the fine grain details so you just want a general overview of the behavior of the application.

ii) If your goal is to do a fine-grain analysis more detail is needed. The other typical approach are the tracing tools. High accurate and detailed information can be obtained, but it needs more resources in terms of disk and analyst effort because more data implies more complexity.

There is not a general best choice but is just about the needs. One disadvantage of profiles is that the high degree of aggregation used to hide variability, for example if there is the case where one single function behaves different depending on the phase of the execution. In this direction research has been drived, e.g. phase-base profiling [Malony et al., 2005] or CCG [Knupfer and Nagel, 2005]. Other one is that profilers discard the temporal dimension so the execution sequence is lost. About the advantages, it is a general summary that allows the user to take a peek about what is going on, e.g. if the user want to speedup a code a profiler is a good starting point to figure out where to center the efforts.

If you look at tracing tools the obvious advantage is that there is a lot of information available, in fact with not a lot of effort profiler information can be derived from traces like is done in Cube [Saviankou et al., 2015]. It can be considered profiler capabilities as a subset of trace analysis capabilities so there should be a drawback for tracing because tracing is not always the correct choice. The main drawback is the scalability for both: i) In tracing time and ii) in analysis time. The problem is becoming worst since the capacity in terms of parallelism is increasing[5] so the number of tasks to monitor makes tracing techniques by one hand hardly scalable and by the other hand tricky to analyze because the huge quantity of data and because the increasing

[4] The program layer bring together the program itself, programming models, frameworks and libraries.

[5] As have been said High performance computers become more complex every generation, e.g. the current number one on the Top500 list is the Sunway TaihuLight with 10,649,600 cores[Top500, 2017] .

response times during the interactive exploration of analysis results. Researchers are currently facing the scalability problems in this two forms. There are currently several specialists driving research in the tracing scalability, they are trying to reduce the overhead of tracing in terms of time and trace sizes by mean of several techniques like machine learning, data-mining and so on like in [Llort Sánchez, 2015] or by on-line compression like in [Noeth et al., 2009]. About analysis time, the complexity of the analysis can be overcome by adding some sort of automatization. Some examples of this research line are automatic performance analysis [Wolf and Mohr, 2003], automatic structure extraction [Casas et al., 2007], phases detection [González García, 2013] fundamental factors models [Casas et al., 2008], automatic analysis throw deep learning [Söderlind, 2017] and so on. Following to this ambition this thesis is focused on the analysis field and is devoted to contribute to ease the work of the analyzers by aid part of the analysis.

1.2 Background

This thesis has been developed in the Performance Tools team at Barcelona Supercomputing Center and therefore the developments explained in this document have been designed to fit in this environment, nevertheless the techniques are general so could be applied to any other environment.

1.2.1 Software

In this section the most important pieces of software used for this thesis purposes are described.

Extrae

Like its name (in spanish) indicates, it is about extracting information. Extrae is the piece of software in charge of inject monitors to the application to be studied and extract valuable information for the analysis. Different mechanism for monitoring are available being the most used the interposition mechanism by means of the `LD_PRELOAD` environment variable present in UNIX systems. By this mechanism just calls to a given dynamic library can be instrumented because what it does is to divert shared library calls by the program to extrae, then extrae get the information and calls the actual library function. It has the advantage that the source code is not needed so it can work directly with already compiled programs without the need of recompile anything. It supports several frameworks: i) MPI (for several implementations) ii) OpenMP iii) OMPSs iv) CUDA v) OpenCL vi) pThreads and virtual machine instrumentation: i) Java ii) and Python. The information that can be collected by Extrae is not just timming what is maybe the most important one but also some hardware counters by means of PAPI interface. All of this information can be complemented by user events that can push the desired information to the tracefile by means of the Extrae API. One possible use of the Extrae API is to extract information about how different iterations from a given loop are behaving. All these gathered information from all threads/processes in the target application is finally merged and presented to the analizer as an ASCII tracefile. Since the task of analyze a ASCII file is a tough work, a visualizer has been developed, that is Paraver.

The methodology presented in this thesis is about post-mortem trace analysis and this traces will be generated by extrae so it is an important piece of the overall workflow.

Paraver

The task of analyze raw numbers is tough so the natural choice is to represent them visually since we, as humans, are very good understanding visual patterns. For example in mathematics is usual to use plots. In the field of performance analysis, visualizer tools also has been developed, having in BSC the Paraver (PARAllel Visualization and Events Representation) tool that as its name indicates (in spanish) is about "to see". Paraver allows to have a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems.

Its power lies on two main concepts. The first one is that paraver traces are semantic agnostics therefore it is provided by auxiliar files called pcf (paraver configuration file[6]) and row[7] (names configuration file). It allows to extend the tool with support for new performance data or programming models easily. The second and more interesting is that the derived calculations from initial metrics from trace is not hardcoded in the tool but configurable so for example to derive a typical metric like IPC you should filter number of instructions by one hand, number of cycles by the other hand and then perfom a division. This powerful approach give the resposability to the user that can leads to the "blank page syndrome"[8] for the not so skiled users. For avoid this sort of problems the Paraver package provides a set of configuration files of the most used views that perform all the filters and derivations for you.

Even if paraver is not the most important piece of this thesis workflow is important to introduce since it is used for the validation part by comparing the results of the proposed tools with the information that paraver provides.

Mercurium

Mercurium[9] is a source-to-source compilation infrastructure aimed at fast prototyping. Current supported languages are C, C++ and Fortran. Mercurium is mainly used in Nanos environment to implement OpenMP but since it is quite extensible it has been used to implement other programming models or compiler transformations, examples include Cell Superscalar, Software Transactional Memory, Distributed Shared Memory or the ACOTES project, just to name a few.

In this thesis it is an important piece since it has been used for the loops characterization (explained in section 3.3.1). The reason is that there were the need of gather information from loops and its iterations . This information was impossible to take by means of typical `LD_PRELOAD` mechanism so the alternative was to instrument user code and fire events to trace with the desired information. Since this work is tough when dealing with large codes, a source-to-source compiler, responsible for doing all the transformations is ideal.

[6] Do not confuse with cfg files

[7] It is called row because it provide names for the rows of the timeline so for the space axis

[8] https://en.wikipedia.org/wiki/Writer%27s_block#Blank_page _syndrome

[9] https://pm.bsc.es/mcxx

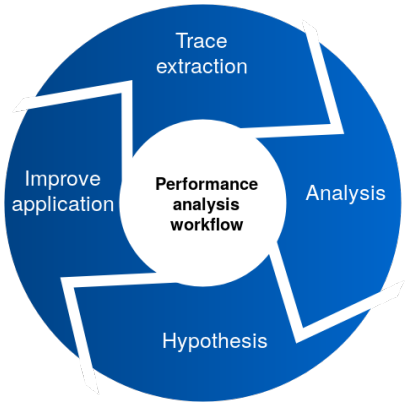


Figure 1.3: Performance analysis workflow

There are some other source-to-sorce compiler infrastructures over there like LVMM[10] or Rose[11] but finally Mercurium have been choosen because advantages derived from the proximity since it is under maintenance and development by the Programming Model team in BSC[12].

For more details about the work done with mercurium go to annex B.

1.2.2 Analysis workflow

In an application-centric approach, the performance analysis is a cyclic process consisting of observing the behaviour of the application so as to hypothesize the possible problems that affect its performance and finally translate these hypotheses to improvements in the application re-starting the cycle to validate them as is depicted in figure 1.3. Obviously, the less number of iterations of this cycle the less time wasted and it directly depends on the quality of the hypothesis that is strongly related with the possibilities the analysis tools provides.

This work is centered on the analysis phase, so lets go further and look at its subphases. When dealing with little and medium size traces, visualizers used to be responsiveness enough to perform analysis directly to the traces but when the size surpass a given threshold several previous steps should be done. In last cases analysis phase is typically subdivided into the following subphases (see figure 1.4):

i) Before filter phase, is used to be impossible to work with visualizers so the main goal is to end up just with the needed information for inspect the structure of the application.

ii) HPC applications are used to have a common idiosyncrasy that is to present very repetitive patterns both in space (different processes) and time dimensions. Exploiting this typical characteristic, next step is cut the more interesting parts of the execution e.g. an iteration or a set of iterations that seems to presents interesting information because its extremely bad performance. This cut is get from the original trace, i.e. with all the original information.

iii) Last and most important, the inspection.

[10] http://llvm.org/
[11] http://rosecompiler.org/
[12] https://pm.bsc.es/

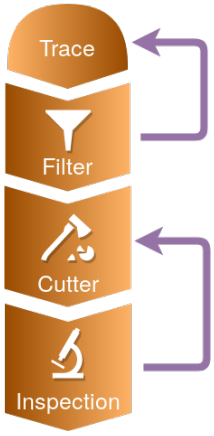


Figure 1.4: Analysis subphases

This work could drive to repeat the same process several times and since every one of the steps potentially have a very time consumption, when dealing with huge trace, it can be a waste of time. Both first and second subphases could need to be repeated several times because the first filter is not good enough or because on the last phase analyzer realize the cut is not as interesting as looked like. The intuition of a high skilled analyst plays an important role in this process.

As has been covered in motivations (section 1.3) this thesis approach can be used for improve the inspection subphase but also to help to the decision or at least give some clues to the analyzer to what region is the most interesting to cut reducing the number of iterations over the subphases of the analysis giving more time to the inspection work.

1.3 Motivations

We can say that HPC applications shares a common idiosyncrasy between them. Mathematics solvers needs to iterate until the result converge and simulation software are used to be programmed to evolve over timesteps. So in general, HPC applications consists on a big outer loop that is being executed the same code but with evolving data (time axis), and same code with different data for every parallel process (spatial axis), i.e. SPMD[13]. Additionally this kind of workloads are used to be burst syncrhonous, i.e. all ranks are executing computation and communication phases in a syncrhonous manner. For these so regular executions a relatively simple representation of the whole execution could be extracted, i.e. the trace could be reduced to a minimum pseudo-code expression with aggregated data for the whole loop so its somehow folding all the iterations space. This pseudo-code expresses indeed the actual structure of the application.

There are several factors that motivates for the extraction of the internal structure of an application being the most important the first one.

[13]Single Program Multiple Data

1.3.1 About improve understandability of execution

Even if performance tools have been demonstrated to be valuable on detection of bottlenecks for a long time, since it demands high skilled profiles for the analysis, developers are not used to use them but delegate this work to the actual specialist. One example is the POP project[14]. What it means is that analyzers are not used to work with their own codes so they are used to be agnostics about the structure of the program and it difficults the process of relate performance metrics to program code. Having the structure of the program will lead to better understand the application by allowing to analyzer to have a mental schema about what the program does and allows to relate differencies in terms on performance to different program phases. This aid on the understandability will lead to better and faster analysis while also drives to better reports to the developer.

There are previous works (discussed widely on section 2.1) that tries to represent the internal structure of an application but they are used to use directed graphs like in EFG [Aguilar et al., 2016] or just callstack trees like in Cube [Saviankou et al., 2015]. The drawback of represent the application structure just with directed graphs is that it is a so simplified view, in the referenced case it is impossible to pinpoint a given loop (directed graph cycle) to the actual code because the lack of callstack information. In case of the referenced approach that uses callstacks tree, the lack of information becomes from the fact that there is no representation for loops. Profilers are used to present syntactic structure with call path hierarchical views where you can easily pinpoint some aggregated behavioral metrics to a point in program. The counterpart is they present the structure in a static way since they lost the temporal dimension. It prevents the analyst to have a clear representation about dynamic execution like loops iterations and they are exposed just indirectly by the number of calls to a given function.

The identification of this lack of clarity on the structure representation motivates as well this thesis. The proposal is to generate a pseudo-code with loop and conditional structures such that the temporal order is maintained. Additionally it can be easily pinpointed to source code and aggregated metrics can be shown so it can be understood as a sort of mix of the two approaches references above.

1.3.2 About identify regions of interest

The typical tools like visualizers are not enough responsiveness when dealing with traces with several GB of information so analysts needs to first preprocess the trace before go ahead with the actual inspection. This preprocess consists basically on filter and cut the trace (widely explained on 1.2.2) for identify and inspect just these interesting hotspots where the bottlenecks seems to be. Automatically extract the structure of the application will give the possibility to agregate information by phases and loops so it can aid to analyzer into the work of find out the hotspots. In this paper [Trahay et al., 2015] they present an automatic tool that extract these points automatically based on the criteria that iterations that behave different from the mean are the interesting ones.

[14]The POP (Performance Optimisation and Productivity) center of excellence in computing applications provides performance optimisation and productivity services for academic and industrial codes in all domains. https://pop-coe.eu/

The problem with the approach presented on the cited paper is that the methodology is difficultly scalable since it presents high complexity and is I/O intensive since the data should be readed repeatedly and it is a problem when dealing with big traces.

1.3.3 About been scalable

Current approaches about extract the intern structure (syntactic but not behavioral, more details in 2.1) based on a post-mortem trace analysis rely on pattern mining techniques (the basic idea of these algorithms are exposed on annex A) that is the obvious choice because a trace is basically a sequence of events sorted by time. The problem with these sort of algorithm is the complexity. So the last motivation is about reducing the complexity of this task. The performance of the proposed algorithms are prohibitive like in [Trahay et al., 2015] [Safyallah and Sartipi, 2006] [López-Cueva et al., 2012] so taking into account the exponential increassing of traces sizes this thesis contribution is the idea to perform intern structure analysis instead of as a pattern mining problem, as a classification problem exploiting by this way the idiosyncrasy of HPC applications. Classification algorithms like clustering presents about quasy-lineal costs.

1.4 Expectations

The goal of this work is to automatically detect the internal structure of an application and correlate it with performance metrics by mean of a post-mortem trace analysis that will help to have a general overview about the phases of the execution. This will allow the analyst to build up a mental schema about the application and understand more easily what could be going on. Furthermore it will contain source code information that will allow to easily pinpoint one metric to the actual code providing an enhancement on the analyst-developer understanding. Additionally this structure extraction can help the analyst to decide what part of the trace analyze first since it will allow to know the loops and iterations boundaries.

The main objective is to detect those sets of events in trace that are being repeated together several times (above a given threshold) and group them into loops and subloops that will represent faithfully the actual structure of the application. This sort of analysis have been done typically by using sequential pattern mining techniques what is in principle the set of techniques that better fits to this problem since a trace is just a sequence of events ordered by time.

Based on previous works and on the premise that HPC applications used to present the same idiosyncrasy (SPMD, burst synchronous,...) has been decided to explore a new way to detect those patters, motivated mainly for the fact that pattern mining techniques are in general computationally expensive. By using data mining clustering techniques we expect to have an scalable solution since it is used to present quasy-lineal complexity with the number of elements to clustering. The mapping between elements to clustering and events on a trace is not bijective but exhaustive, so every element potentially represents several trace events, what is in fact the same instruction presenting its dynamic aspect because the execution of loops. Assuming repetitive executions, the number of elements to clustering would be about constant despite the increasing of the execution time.

In this work is not expected to end up with a completely functional and performance refined tool but demonstrate how this approach can present valuable results with a good scalability on increasing trace sizes.

Chapter 2

State of the Art review

On previous approaches that shares similar motivations with this thesis and a discussion about them.

2.1 Previous and related work

THE analyzed literature can be split into two main subsets. By one hand we have the syntactic structure analysis tools that provide information about the actual program structure and by the other hand the behavioral structure analysis tools that want to expose the different phases on an execution in terms of performance. The former subset of works is the most related to our goals even if not all of them are using the structure detection for improve analysis but also for trace compression. The last subset is not so related but it have been considered interesting to analyze them since provide valuable insights for our purposes.

2.1.1 Syntactic structure

This section has been divided into two subsections. The first is about algorithms used on an on-line structure analysis, i.e. Overall trace is not available from the beginning so the structure is being detected while the application is being executed. The works presented in this sections are mainly concerned about reduce the size of the trace by compressing these repetitive parts so they are facing the tracing scalability problem. The second section is about off-line structure analysis, that unlike before trace is available so a trace post-mortem analysis is done.

On-line structure analysis

In [Noeth et al., 2009] they are concerning about the scalability of the tracing part. They claim reductions of about a thousand in terms of trace size just by detecting the structure of the application, e.g. If the same thing is repeated 100 times, just saving it once and tagging with the number of times it is executed should be enough. They propose to use RSD (Regular Section Descriptors) to express MPI events nested inside a loop and a sequence analysis algorithm for detect the repetitive patterns. Their compression is done in two phases. The first one is an intra-node

compression, where the repetitive patterns arise and the second one is inter-node merge, where all single-node compressions are merged forming the whole application trace. They maintain two sequences the "target" that contains the already detected sequences sets and the "match" sequence that is formed by the newly acquired trace records. The compression algorithm maintains a queue of MPI events and attempts to greedily compress the first matching by in four steps procedure:

i) Head and tail of the match sequence is determined by traversing the queue backwards such that the last item is the tail named "target tail" and the next coincident item with this tail is the "match tail" so just the previous one will be the "target head".

ii) Following from "match tail" find out the item that is equal as "target head". This will be the "match head".

iii) An element-wise comparisson is performed in order to check out whether both sequences match or not.

iv) If there is a match the "match" sequence is merged to the "target" and is removed from the queue.

Also they claim that even if their main target is to compress traces, Scalatrace also can be used for analyze the application structure by doing a little demo showing how it can detect the most outer loop iterations or timesteps. One of the main drawbacks of this approach is that the complexity of intra-node compression can be of $O(n^2)$ nevertheless they are avoiding this by limiting the algorithm search (first step) with a windows size. They claim that with a windows of 500 events is used to be enough. Additionally they introduce useful concepts as:

i) Calling sequence identification used for unambiguously identify different MPI calls that lie on different code positions. It is an important matter because if we do not take into account we would end up having aliasing problems, different calls would seems the same.

ii) Recursion folding signatures for dealing with recursion. If an MPI call is called recursively, the signature based on the callstack would identify same call as different. The proposal is to fold the same call on the call stack, e.g. $A \rightarrow A \rightarrow A \rightarrow MPI$ will be $A \rightarrow MPI$.

In [Aguilar et al., 2014] they present the Event Flow Graphs (EFG). EFG are weighted directed graphs where every node is an MPI call and edges the transitions between them being the weight the number of transitions done from one node to the other, so the program code blocks executed between them. Graph nodes can contain aggregate information like call duration or message size and edges can be attached with information about CPU burst like performance metrics like IPC. The EFG is constructed like in Scalatrace at monitoring time and it consists on two basic actions: i) Every time an MPI call is detected gather all information and store it in a hashmap indexed by the MPI call signature. The signature is a k-tuple of components which represents relevant metrics like MPI call type and source code position. Every entry of the hashmap is directly related with one node. ii) Also on every MPI call detection a transition from one node to other has to be stored, it is what they called "signature history" and it consists on a set of pairs $(signature_{i-1}, signature_i)$

and for every pair an scalar value is also stored that indicates number of times this transition is taken. This set of transitions are the edges of the EFG.

So far no information about order is taken into account so additionally they present in [Aguilar et al., 2016] temporal-EFG that introduce more information for these cases where the execution order can not be reconstructed with the previous EFG. They claim this technique can be used for trace compression, application structure detection and visual performance analysis. Following with application structure detection, what is where this thesis is focused on, they use algorithms for cycle detection over the t-EFG (DFS-based) and once cycles are detected the graph is transformed to a hierarchical tree where loops and subloops are showed up. Statistics about loops can be gathered like number of iterations, total time in loops and so on.

Offline structure analysis

Some of works in this section are out of the HPC field, this is because the structure extraction is also useful for other purposes like reverse engineering on interactive software nevertheless they are actually useful since the objective is quite the same.

Starting with [Safyallah and Sartipi, 2006] they propose analyze execution traces of software systems in order to extract the intern structure for improving reverse engineering process. For that end they first instrument the application (the entry/exit of functions) and a set of relevant task scenarios (actions) are selected, that examine a single software feature, called feature-specific scenario set. These scenarios are executed and traces are collected. Second step is the execution pattern analysis that extract both, intra-scenario set where patterns that are specific to a single software feature and inter-scenario set where more general patterns appears, i.e. no feature specific structure so the main structure of the application. For pattern analysis they rely on sequential pattern mining techniques using a modified version from [Agrawal and Srikant, 1995] what is a slightly different from what has been explained on A.4 and is also using candidates generation approach. They are able to find inter and intra-scenario patterns by tuning the Minimum support such that for detect feature-specific patterns it is reduced to about 5% and to find out inter-scenario patterns it is incremented to about 25%.

Similarly in [Zhao et al., 2008] they present an approach to extract the intern structure of software systems, mainly interactive, but using graph-based substructure mining algorithms instead of sequential based mining techniques. They propose a four step methodology: i) Trace collection ii) Trace preprocessing iii) Grammar induction iv) and Grammar parsing. Traces have several information but the most important is the enter and exit from methods that allows to derive the call-graphs. These call-graphs are saved as a linked list of caller-callee relations. Following, to facilitate the grammar induction step, the data is simplified by remove some repetitive and fine-grain details such that low-level methods. The result of trace preprocessing feeds the grammar induction step that rely on VEGGIE (Visual Environment for Graph Grammars: Induction and Engineering). Induction algorithm iteratively finds common substructures from a given set of data, and organizes the hidden hierarchical substructures in a gramatical way. When a common frequent substructure is found, a grammar production will be created.

This paper considers to use a different approach from sequential pattern mining and it is also interesting since the last seems is the dominant approach. They show up some results being maybe

acceptable for the targeted analysis but not for this thesis objectives since they report execution times of about 70 seconds for traces with about 90 events.

In a bit different scenario, [López-Cueva et al., 2012] they talk about debugging and optimization process of software for SoCs[1] by means of traces post-mortem analysis. They explain the complexity of SoCs drives the analysis of these traces difficult because the high quantity of information that they are used to collect so they are facing the problem about scalability on analysis. They argue that the manual analysis of execution traces is becoming an unmanageable task so this task has to be aided by automatically extract pertinent information, what is in fact the structure of the application, and for that they rely on pattern mining techniques, specifically frequent periodic pattern mining. As has been explained in section A this sort of algorithms are used to work with set of transactions so in order to adapt trace mining to that algorithms they have chosen to split the trace into a set of subtraces (by time intervals or by function name). They say *"we are interested in discovering sets of events that occur periodically [...] but the order is not taken into account [...] the order can change according to the scheduler (in a multi-thread environment)"*. It remembers to the algorithm explained at section A.4 in the sense that the events in windows were not assessed to be in order since they could happen in parallel. Furthermore the split of the trace into transactions remembers to the windows explained in that same section. Additionally to the classical temporal sequence mining they introduce the definition of cycles as *"When an itemset occurs over a set of transactions and the distance between any two consecutive transactions is constant"* and periodic pattern as *"a set of consecutive cycles over the same itemset and the same period"*.

The mining consists on a four step algorithm where the first one is the responsible of find out all the cycles of all possible periods (from 1 to #transactions / min. support) containing a selected item. The rest of steps are responsible to refine the output in order to end up with the minimum useful information. First step is intuitively slow and generates a highly redundant information so is hardly scalable.

Returning to the HPC field, in [Trahay et al., 2015] they present an approach for select points of interest automatically from an execution trace, understanding as points of interest these iterations that behave different from the majority. The first phase is a post-mortem analysis of a given trace. This analysis is about finding patterns of events that are repeated. Their algorithm is about finding short repeated sequences of events and try to expand the pattern. After detect the intern structure of an application, the analysis of durations of the different iterations of the detected loops is an arbitrary construction, once done, they filter all iterations that behave similar and expose to analyst these iterations that are outliers assuming these are the most interesting ones.

They propose a iterative three steps algorithm:

i) Find out a sequence of two consecutive events that appears several times. All the subsequences are then replaced with a pattern construct p_1.

ii) Next step is about finding loops composed of p_1. This is done by comparing every p_1 with next event, if they are equal then both are grouped into a loop.

[1] System on Chip

iii) Last step is about expanding the pattern p_1 by looking at following event. If all p_1 have the same following event then it is integrated, if several of them shares the same following event, new pattern p_2 is created otherwise the pattern can not be expanded.

Steps 2 and 3 are repeated until no more expansions can be done. Then the process starts again from step 1 until no more pairs can be found.

This algorithms can be classified as a pattern growing algorithm (described in section A.3) but without the projection step so all data is traversed again and again. They said their algorithm is dominated by the first step that presents a complexity of $O(n^2)$ with the length of the patterns.

Compressed Complete Call Graphs (cCCG) was presented by [Knupfer and Nagel, 2005]. It can be said it is about profilizing a trace. It consist on finding repetitive patterns for loosely or lossless compression. CCG is basically a graph of function calls of a program so the main structure is defined by the function call hierarchy while additional information are appended usually as leaf nodes. The construction phase is quite simple. The trace is traversed in a sequential manner, every time a function enter event is detected, new node is generated and append to the current active node. The other way around when exit function event is read, the current active node is finalized and all information concerning to this node is presented e.g. duration. Additionally, while constructing, the graph is compressed. The basic idea is to replace n repeated sub-trees that are equal or similar with a reference to a single instance saving $n-1$ remaining copies. For similar is understood that when comparing nodes not all properties must be equal for example is not needed some scalar values like duration match perfectly (some configured deviation is acceptable) but properties like function id must. They claim compression ratio about 200 can be achieved with this approach. This approach improves profiling in the sense that same function with same call path and different time behaviour is exposed to analyst but still presents a lack of information on the order of the execution of the different graph paths so application structure is exposed just partially.

2.1.2 Behavioral structure

Consider the same bunch of code will behave in the same manner in general, it could be assumed that there is a powerful correlation between the behavior and the code structures. From which can be said a behavioral analysis is a side-channel analysis because indirectly the internal syntactic structure can be betrayed. This consideration is incorrect in general so the main goal of the different approaches presented in this section is not to present a syntactic but a behavioral structure. The motivation is that when analyst deals with syntactic structure, like in profilers, time variations of the same functions is hidden, this situation can appears for example when calling same function with different parameters. For the analyst point of view could be more interesting to have this information unfolded. Take into account that this same property can end up identifying different parts of the code as the same phase.

Even if the goal of the approaches explained below does not match with the goal of this thesis, they provide a really useful insights as a related work.

In [Casas et al., 2007] they propose automatically extract the internal structure of an MPI application from a Paraver tracefile and provide to analyst just representative phases and they

rely on signal analysis for this propose. Their analysis consists on two main steps: i) The first is to clean-up the trace by identify the perturbed regions. Perturbed regions are those parts of the trace that has been perturbed nor by the application nor by architecture but by external factors such that unknown system activity or tracing package. Their clean-up phase is centered on remove noise from tracing package, i.e. flush to disk. By building up a signal based on flushing events and transform it by Closing morphological filter they end up with this perturbed phases. ii) The second step is the identification of the phases. It is done by means of autocorrelation and periodicity analysis of a signal. This signal is build up from any metric like instantaneous FLOPs but they use number of MPI point to point calls being executed. Once the period is successfully detected the same process is done recursively on one of the periods. This allows to have a hierarchical structure. Finally the output is basically information about the different phases like number of iteration and timing plus some representative cuts of the original trace.

In [González García, 2013] propose a technique to identify the different execution phases using clustering techniques. The units to be cluster are segments of the execution to be defined, i.e. execution bursts that takes place between a given events such as MPI functions or events fired by the user to trace. The first step is about reducing the complexity of the clustering by filtering the less relevant of these computation burst, i.e. little bursts according with a given threshold. The second step is the clustering itself. These defined execution burst will have behavioral information attached such as time or hardware counters that will be used for cluster them in a user defined N-dimensional space. In order to reduce the dimensionality the typical approach is to use two different dimension sets: i) Completed Instructions against IPC. This configuration provides a performance view. ii) The second is Completed instructions against L1 and L2 cache misses. This combination reflects the impact of the architecture on the application structure. Once the clustering is done, this information can be sent back to the trace and can be visualized by the analyst. Additionally an interesting analysis can be done with the shape of clusters like for example, working with IPC vs. Number of instructions dimensions if the shape of a given cluster is flat on the second axis it means that there is an imbalance on instructions. Clustering in the field of performance analysis was used before this proposal just for classify processes that behaves similar so its utilization by identifying different phases on temporal dimension opened the door to new research paths.

2.2 Discussion

About syntactic structure literature, a clear limitation of the first approach presented in [Noeth et al., 2009] is the complexity is about $O(n^2)$ and for that reason they propose the use of a windows that will limit the search. Even if they are preventing for this high complexity it has a lateral effect since they are limiting the recognition of large patterns. In [Aguilar et al., 2016] they propose to use graphs for application structure inspection. After collect and build up the tEFG, they apply a DFS-analysis in order to give to the user a hierarchical graph where the structure is revealed but what they no say (and it seems to be because the figure they show) is that this derived graph there is a lack of temporal information, so it can be considered a limitation of the methodology (also we have to say that this is not the main goal of the proposal). From both papers there is a valuable information (most in the second one) about the idiosyncrasy of HPC applications in

what this thesis rely. [Noeth et al., 2009] say *"[...] the outermost loop of the code that contained repeated MPI calls. This timestep loop is of particular interest for performance modeling as convergence algorithms are often based on either fixed iteration bounds for the number of timesteps or epsilon-based error constraints resulting in input-specific number of timesteps. [...]"* and in [Aguilar et al., 2016] they talk about the "big outer loop hypothesis". Additionally says *"[...] by detecting the graph cycles (nodes are MPI calls), we are detecting the actual loops that drive the simulation in the application"*. What it means is that monitoring the MPI library is enough for having a clue about the general structure of the application, they acts as the fundamental pillars.

In off-line syntactic structure algorithms, that is in fact the approaches what shares more things with our objectives. The dominant algorithms here are the sequential pattern mining algorithms with variations for every case. Because the importance of these algorithms for this field, a previous study of them have been driven. If you want more details go to annex A. Even if the structure of the applications can be betrayed successfully, the sequential pattern mining, particularly for candidate generation-and-test techniques, and when low support is used, performance can degrade dramatically. In apriori algorithm presented in A.2 have been observed that the number of candidates needs to be generated for a relatively long itemsets is really huge and presents an exponential growth with the number of items in the itemset. Additionally for every one of these candidates, data needs to be traversed again on the test step.

If we look closer, one can realize that the complexity of the algorithm mostly depends on the expectations of the complexity of the betrayed patterns. About studied literature in syntactic structure [Safyallah and Sartipi, 2006] [Zhao et al., 2008] talks about get the application structure at level of entry/exit functions, and as they talk it can leads to a high complex structures that needs post-process, for them pattern mining algorithms is the natural choice. In [Trahay et al., 2015] they talk about find patterns of events that are both, MPI calls and function calls in general and following the work of paragraph above, rely on pattern mining. [Noeth et al., 2009] also use some sort of sequential pattern mining but the feeling is that their needs drivers to a more simple algorithm. Finally in [Aguilar et al., 2016] they just perform a directed graph building because they just get MPI calls information, so simpler patterns will be detected. Strong reason for prefer a simplified analysis for the last two is they are working online, so they are trying to decrease the overhead as much as possible.

About last set of proposals, those classified as behavioral structure approaches (in 2.1.2), it can be seen signal processing and general data mining techniques are used. The reason is because the behavior is expressed with scalar values like number of instructions, cycles, cache misses, etc instead of by a sequence of a defined set of events like MPI calls so sequences pattern mining is not the best choice. About [Casas et al., 2007] it shares with this thesis the motivation explained in 1.3.2. They can present a hierarchical graph with detected structure but without temporal order information what is a weak spot. About [González García, 2013] it brings the idea of the use of clustering for structure detection, using burst clustering with hardware metrics you can have aliasing in the sense that two burst that execute different parts of the code can be behave similar and lie on same cluster but with other metrics, actual code structure can be betrayed as is demonstrated in this thesis.

Chapter 3

Scalable syntactic structure detection

On proposed methodology basis and further studies

3.1 Application structure by classification

TAKING into account the motivations, exposed in 1.3, and the previous works studied and discussed in 2.1 the idea to introduce a new approach arise.

Our intention is just use information from MPI calls, mainly for three reasons: i) It does not needs to instrument the source code of the target application so it can exploit the `LD_PRELOAD` capabilities of extrae. It is a big deal since the source code is not always available to the analyst. ii) trace size is small compared with traces with more information like entry/exit from functions used on other approaches iii) and as it has been argued in 2.2 monitoring the MPI calls is enough for having a clue about the general structure of the application.

The proposal presented in this thesis is to explore a new way to detect the syntactic structure of an application by applying clustering. The essential point is that the problem has been converted into a classification problem so instead of looking for these events that are being repeated several times during the execution, close enough one each other, let's face the problem of arrange the MPI calls that belongs to the same loop to the same cluster. Then can analyze the hierarchical arrangement of these clusters to detect nesting between loops. Once done, the pseudo-code representation construction becomes quite easy. The scalability of this method resides on the fact that HPC applications are strongly repetitive over the time, so the number of unique MPI calls to cluster remains about constant despite the growth of the execution time, additionally clustering algorithm such that K-means presents an attractive linear complexity and DBSCAN a quasi-lineal in general.

The key point of the presented method is to cluster MPI calls into loops so we have to figure out what loop feature or features are able to identify them in a better way. As have been said we only work with mpi calls so there is not actual information about loops available in trace, instead of that we perform a side-channel analysis by using mpi calls and the computation between them as a proxy of iterations information such that for example, number of iterations can be understood as number of times a given mpi call is being executed. From this definition it can be understood

that the only loops this proposal is going to detect are those loops that contains mpi calls.

The features that we are going to use are:

i) Number of dynamic iterations

ii) Iterations mean time

About the first, we rely on dynamic number of iterations. Iterations of a given loop does not depends only on the number of iterations in its definition but also on the number of iterations on the definition of the parent loops such that if there are two loops having n and n iterations being one nested on the other, in the dynamic domain, they will have n and n^2 iterations allowing to differentiate them. So by this first feature an arbitrary number of nested loops can be identified. Number of iterations will corresponds to the number of repetitions of a given mpi call. The situation where two loops lies on the same nesting level and have the same number of iterations will drive to identify both with same identification if we just rely on number of iterations. This situation is less probable but since it has been detected in some cases it has to be covered. Luckily even two loops have the same number of iterations they use to perform different work. The way to measure the quantity of work a loop does is by means of the iterations mean time, what is the second selected metric. We consider the iteration time, to be the times that elapses between two executions of the same mpi call.

3.2 Proposed methodology

In our methodology we rely on the observation of the MPI calls to infer the fundamental internal structure of the application, the reason is that the principal loops that drives the execution on HPC applications used to contain the MPI calls needed to perform the communications between the different processes, so looking at them should be enough for an overview of the structure in most cases. The proposal consists on a three fundamental pipelined steps: i) Trace reduction: It consist on the trace parsing, aggregation and derivation of the metrics related to the MPI calls. ii) Loops clustering: This step is where the gathered MPI calls are clustered and every one of the resulting clusters are considered as different loops. iii) Loops merge: Once the calls are grouped into loops, the relationships between these loops have to be studied such that the actual structure of the application in terms of nesting relations is showed up. iv) Inter-rank reduction: Where same calls from different mpi ranks are merged. v) Pseudocode construction: Final step is about building up a representation of the detected structure such a way it ease the interpretation of data. The chosen format have been pseudo-code with attached performance data. In figure 3.1 it can be seen an overview of the explained architecture.

3.2.1 Trace reduction

This phase is a sort of pre-process of data. Data is presented as a tracefile that is basically a sequence of timestamped events and it is transformed to a set of MPI calls with attached information that will be the input for the next phase, i.e. the clustering. This phase performs two actions, i) Reduction ii) and aggregation & derivation

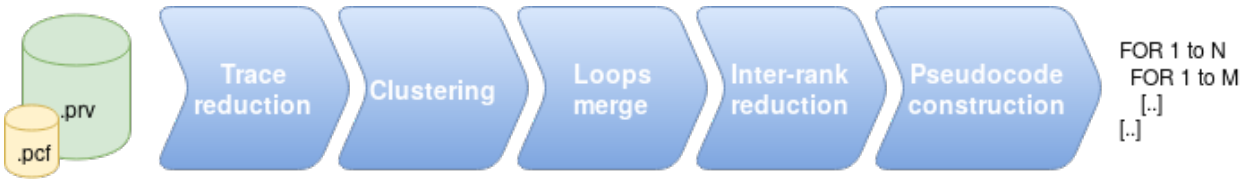


Figure 3.1: Methodology workflow diagram

About the former, the reduction consists on collapse all the same MPI calls that are sparsed among all the time axis. Very inspired on [Noeth et al., 2009] two MPI calls results to be equal if and only if have the same signature. In our case the signature is defined by the entire callstack, i.e. a sequence of pairs $(file, line)$ that unambiguously will define the dynamic position in code of a given MPI call. Additionally contains the rank-id what means that even if two mpi calls have the same call path if they have been executed by different processes will not be considered the same.

We can define T as the sequence of mpi calls ordered by time and $|T|$ the total number of mpi calls that is strongly related with the size of the input trace. Now, $c \in T$ is an MPI call being $t(c)$ the entry instant on this call and $signature(c)$ its signature. Having Ω the set of reduced c, then $r : T \to \Omega$ is the reduction function. This is an exhaustive function such that $\forall c \in T \, \exists \omega \in \Omega : r(c) = \omega$ and fulfills $\forall x, y \in T : signature(x) = signature(y) \Leftrightarrow r(x) = r(y)$. Also ω elements collect all times from reduced calls such that $\forall c \in T \, \exists \omega \in \Omega : r(c) = \omega \Leftrightarrow t(c) \in T(\omega)$ being $T(\omega)$ an ordered list of times. Additionally elements in T could have some features $\lambda \in \Lambda$ attached that becomes to ordered list on Ω space similar to $T(\omega)$.

Once the reduction is finished next step is to perform the aggregation and derivation. Aggregation consists on aggregate the features belonging to those scattered calls, generally the arithmetic mean and standard deviation on for example the size of the messages, duration of the call and so on. This process is not done at reduction time because some metrics needs the entire series like for example the standard deviation. Derivation consist on extract the need information that is not explicit in trace so needs some sort of calculation. On previous section it has been introduced that number of repetitions and mean time between repetitions are the used features to classify mpi calls. Since the number of a given mpi call is repeated has been already calculated in reduction phase, the data to be derived is the iteration mean time that is represented by performing a mean over all elapsed times between two consecutive calls to the same mpi call. The information that is obtained from trace are the timestamps so mean time between repetitions (expresed as $imt(\omega)$) is calculated with formula 3.1 (and similarly with other features $\lambda \in \Lambda$).

$$\forall \omega \in \Omega : imt(\omega) = \frac{\sum_{i=1}^{|T(\omega)|-1} t_{i+1} - t_i}{|T(\omega)| - 1} \tag{3.1}$$

Once the the entire phase is finished, the information that have been stored for every unique mpi is:

i) Number of repetitions and mean time between repetitions: This will be the used data for the clustering

ii) The entire call path and mpi call identification: These data have been used for the reduction but it is still needed for the pseudo-code representation

iii) Performance information of just previous cpu burst: This information is used to enrich the outputted pseudo-code with performance information about the computational phases between mpi calls.

iv) Some configurable hardware counters not just from the previous cpu burst but from all cpu burst between two executions of the same mpi call.

v) All timestamps are also stored for future analysis.

The algorithm developed for this first step is quite intuitive. It basically consists on traverse the tracefile sequentially and every time an MPI call is detected, all the needed information is gathered. Then whether a previous MPI call with the same signature exists is checked out, if no it is added to the Ω set but if yes it is merged with the already existing. Even if the fundamental idea is quite simple, implementing it, in a relatively efficient way, for paraver traces is a little bit tricky because two factors.

i) Communication information for p2p operations like size or partner are different events from MPI call events so communications have to be matched with the actual calls

ii) and extrae does not fire the absolute number for hardware counters but a delta, i.e. Every time an event is fired to trace it resets the every hardware counter. The reason is because the relative value use to be more useful for the analysis. It means that for example to calculate number of instructions from one instance of an MPI call to the consecutive is not as direct as calculate the difference.

Lets define $\Sigma = T \cup \Psi \cup \Delta$ being Ψ the set with all communications in execution and Δ all the hardware counters values fired to trace. In pseudocode 3.2.1 it can be seen the developed algorithm that deals with the explained characteristics of the paraver trace format. In order to deal with communications and hardware counters that lies on different events that the MPI call events, there are a set of buffers. In case of communications (buffers S and D) they are used in case when communication arrives, the calls have not arrive yet. In case of hardware counters (buffer C) is for keep track of values even if they are multiple times set to zero by other mpi call events.

This algorithm presents a linear time complexity (all search in sets are constant since they have been implemented with hashmaps) $\Theta(|\Sigma|)$ that is basically with the size of the tracefile. Only this serial version have been developed but there is a lot of space for improvement since the characteristics of this problem allows to face it with parallel codes. Trace could be split into several files and perform an algorithm similar to the proposed on every one of the partitions, once done a reduction among the different results should be done. Some considerations would be taken into account like for example the position to do the splits or if it would be need all the communications in a single file instead of scattered among all of them. Because time restrictions have been decided to move this sort of considerations to future work.

Reduction step is the key point for the scalability of the methodology since $|T| >> |\Omega|$ because the executions used to present a lot of repetitions so in next step, the clustering is done over a very reduced version of the input trace.

Algorithm 3.2.1: REDUCTION ALGORITHM(Σ)

$S \leftarrow \emptyset$ **comment:** Set of not matched communications on source

$D \leftarrow \emptyset$ **comment:** Set of not matched communications on destination

$C \leftarrow \emptyset$ **comment:** Collection of counters identified by mpi signature

for all $e \in \Sigma$

do
- **if** $e \in \Psi$
  - **then**
    - **if** $origin(e) \in \Omega$
      - **then** $update(origin(e), e)$
      - **else** $S \leftarrow S \cup e$
    - **if** $destination(e) \in \Omega$
      - **then** $update(destination(e), e)$
      - **else** $D \leftarrow S \cup e$
  - **else if** $e \in \Delta$
    - **then**
      - **for all** $c \in C$
        - **do** $value(c) = value(c) + e$
  - **else if** $e \in T$
    - **then**
      - **if** $\exists s \in S : origin(s) = e$
        - **then**
          - $update(e, s)$
          - $S \leftarrow S - s$
      - **if** $\exists d \in D : destination(e) = d$
        - **then**
          - $update(e, d)$
          - $D \leftarrow D - d$
      - **if** $\exists c \in C : signature(c) = signature(e)$
        - **then**
          - $updatehwc(e, value(c))$
          - $value(c) \leftarrow 0$
        - **else**
          - $updatehwc(e, 0)$
          - $C \leftarrow (signature(e), 0)$
      - **if** $\exists \omega \in \Omega : signature(\omega) = signature(e)$
        - **then** $merge(\omega, e)$
        - **else** $\Omega \leftarrow \Omega \cup e$

return (Ω)

Additionally, in order to reduce even more the items to cluster for the next step a filtering can be done. This filter consist on remove these mpi calls that represents the execution of short loops. The filtering use the value that can be calculated by the formula in equation 3.2 what

fulfills $0 < \delta <= 1$.

$$\delta(v) = \frac{it(call) * imt(call)}{T_{exe}} \tag{3.2}$$

Being $it(call)$ the number of repetitions and $imt(call)$ the mean time between repetitions.

3.2.2 Loops clustering

This is the key step of the this thesis proposal because the quality of the output will directly depend on the quality of the clustering. As has been introduced before, the goal is to cluster all elements in Ω into groups that will be considered the loops. There have been a previous discussion about what features to use in section 3.1 so this is not going to be discussed again, in fact this phase is enough general to use any set of features so the intention is being improving the model whenever new and better features will be found for this clustering purposes. Two main trends on clustering algorithms have been considered [Rokach and Maimon, 2005] i) partitioning ii) and density-based algorithms.

About the former, partitioning methods relocate instances by moving them from one cluster to another starting from an initial partitioning. One of the most common criteria for this relocation is the sum of square error (SSE), defined by the euclidean distances between every instance and the center of its cluster, that is trying to be minimized. SSE may be globally optimized by exhaustively enumerating all partitions, which is very time-consuming, or by giving an approximate solution using heuristics what is the most used solution. The most commonly used algorithm following this method is the well-known K-means algorithm. Its complexity is $O(T * K * m * N)$ being T number of iterations of the algorithm, K number of clusters, m number of instances, so $|\Omega|$, and N number of features per instance. This algorithm is quite attractive because its linear complexity but have an important drawback, it needs the number of clusters K in advance that is not trivial when no prior knowledge is available.

About the second, Density-based methods assume that the points that belongs to each cluster are drawn from a specific probability distribution. The overall distribution of data is assumed to be a mixture of several distributions. These methods are designed for discovering clusters of arbitrary shape. The idea is to continue growing the given cluster as long as the density in the neighborhood exceeds some threshold. A particular algorithm that follows these ideas is the Density-based scan that discovers clusters of arbitrary shapes and is efficient for large spatial databases. The advantage respect K-means is that it does not need the previous specification of the number of clusters but needs other two parameters: i) *eps* that is the radius distance needed to consider one instance neighbor or not ii) and *minPts* that is the minimum number of points required to form a dense region. Its complexity is in the general case $\theta(nlog(n))$ with number of instances even if it can degrade to $O(n^2)$ when all instances are retrieved when looking for neighbors for every instance scan. It could happen when instances are really close one each other of because a bad decision on the *eps* parameter.

DBSCAN is the most suitable algorithm for this step because the number of clusters can not be a-priori known, so K-means should be discarded for this step.

Next discussion is about how to set up the two parameters that it needs in order to end up

with optimal results. About the $minPts$ it allows to set when to decide whether an accumulation of instances is considered as a cluster or not, it is a threshold that says how many points as minimum a cluster must have. In our case, every one of the points are mpi calls, what it means is that discarding any of them we are loosing information about the application structure so no one should be discarded. Imagine a loop where there is a single mpi call, if $minPts$ is above 1, it will be discarded and it will not be shown to the user. In conclusion: $minPts = 1$. In the case of eps is not as easy. It has been set to a value that empirically have demonstrated is working well for the majority of the cases. This value is 0.2 in general. They key point here is that all data in all dimensions of the clustering have been previously normalized, so eps is about relative distances between instances instead of absolute.

We can define the clustering function as $c : \Omega \to \Upsilon$ such that $\forall \omega \in \Omega \, \exists \upsilon \in \Upsilon : c(\omega) = \upsilon$. Being Υ the loops set. $|\Upsilon|$ is upper bounded by $|\Omega|$ because in the worst case every loop will have just one mpi call, but the general case is $|\Omega| > |\Upsilon|$ so it is also contributing to decrease the complexity of the next phase that is the loops merge.

3.2.3 Loops merge

Since now we were concerned about what loops do we have on the trace, that are indeed pieces of the overall puzzle of the structure of the application but not the structure by itself. This step is about rearrange these pieces in a way that actual structure of the application emerge explaining the hierarchical relations between the different loops.

One loop have a relationship of subloop with other one, when the first lies in the body of the second. Having $a, b \in \Upsilon$ a is subloop of b is represented as $a \mapsto b$ and fulfills

$$\forall a, b \in \Upsilon, c \in \mathbb{N} : a \mapsto b \Rightarrow it(a) = c * it(b), c > 1 \wedge imt(a) < imt(b) \tag{3.3}$$

being $it(\upsilon)$ the number of iterations and $imt(\upsilon)$ the iteration mean time in the general case. When this relation is done, we talk about merging a to b. Note that this is not a double implication and this is because we can have two loops fulfilling these conditions but belonging to different phases of the execution, so without any relationship among them. What empirically we have seen is that most applications use to have an initialization phase, a body, where the actual application is working and sometimes a finalization phase. One mechanism to differentiate them is looking at loops time boundaries. Additionally since the body is always larger than the other phases (assuming large enough input sizes) another mechanism to differentiate them is to check out how much time of the application every loop can explain.

A naive first algorithm was to sort all loops, by number of iterations in a descending fashion. Then get the first loop in the list an try to merge with the next, if not possible, try with the other one. Note that this algorithm have a complexity in the worst case of $O(\frac{n^2+n}{2}) \approx O(n^2)$ with $|\Upsilon|$. In order to improve it we decided to shrink the search space by classify loops by how many time of the application can explain since nested loops will share this value with parent loops. Remember that this value has been used before for filter mpi calls before the clustering (equation 3.2) so all mpi calls already have it. At cluster level it can be calculated as the mean of the deltas of all mpi calls.

A new DBSCAN is performed over the set of loops and every one of the clusters are the different phases of the execution. Now, the search space is just among loops that forms part of the same phase. The same algorithm is now executed on every phase.

This concept of different phases can be better understood from a geometrical point of view. Having the program depicted in pseudocode 3.2.2 if we plot the already reduced mpi calls in dimensions number of iterations vs. iteration mean time we will have the plot in figure 3.2.

Algorithm 3.2.2: APP EXAMPLE()

```
comment: Short initialization
for 1 to 10
do { for 1 to 2
     do { for 1 to 2
          do { someComms()
          MPI_Call
     MPI_Call
comment: Body of execution
for 1 to 100
do { for 1 to 2
     do { for 1 to 2
          do { someComms()
          MPI_Call
     MPI_Call
```

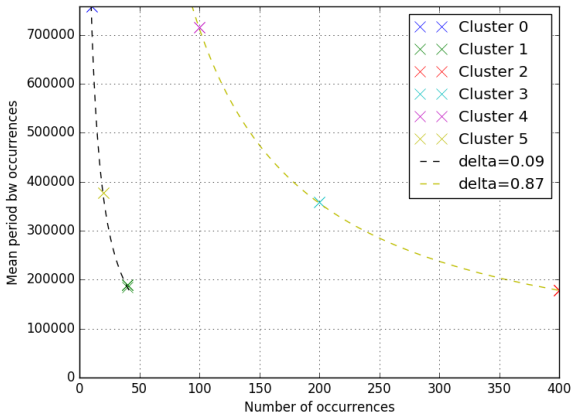


Figure 3.2: Geometrical representation of delta classification

As you can see on figure above, six loops have been detected (this clustering is related with the previous step not with this last explained delta clustering) and classified into two delta groups. The two groups are depicted by these two, black and yellow curves. So one cluster belongs to one delta if it lies over the curve. These curves are described by the function:

$$f(x) = \frac{\delta * T_{exe}}{x} \text{ being } 0 < \delta \leq 1 \tag{3.4}$$

Value of δ is always upper bounder per 1. It is obvious since one loop can not have been executed during more time that the entire application time. Also is lower bounded by zero because if one loop duration is 0 obviously is would mean it has not been executed.

Additionally for simplify the loops merging, delta also can be used for filtering low representative loops. This functionality can be applied even just after the construction of the Ω set (reduction step) since at that point we already have the enough information for calculate δ. Normally, the lower bound value is set to $0,1$ i.e. It is filtering all loops that represents less than the 10% of the overall execution.

On pseudocode 3.2.3 there is the actual implementation of this step. At the very beginning it can be seen how the grouping of the different loops by deltas is performed, now δ are subsets of

Υ, being Δ the set of all δ.

Algorithm 3.2.3: LOOPS MERGE STEP(Υ)

$\Delta \leftarrow deltaClassification(\Upsilon)$
for all $\delta \in \Delta$
do
- **comment:** Sort by it(υ) desc
- $sort(\upsilon \in \delta)$
- **for** $i \in [0, |\delta| - 1)$
  - **do**
    - **for** $j \in [i + 1, |\delta|)$
      - **do**
        - **if** $isSubloop(\upsilon_i, \upsilon j)$
          - **then** $\upsilon_i \mapsto \upsilon_j$

The key point of this algorithm is the *isSubloop* function. Imagine that we have one set of three loops a, b and c with same δ with 100, 50 and 10 iterations respectively. All three can be related in a some way or not. There are six options: i) $a \mapsto b \mapsto c$ ii) $a \mapsto c$ and $b \mapsto c$ iii) $a \mapsto b$ and c is alone iv) $a \mapsto c$ and b is alone v) $b \mapsto c$ and a is alone vi) all three have no relation. You can see the same example where the second option is the correct on pseudocode 3.2.4 and figure 3.3.

Algorithm 3.2.4: APP EXAMPLE 2()

for 1 to 10
do
- **for** 1 to 5
  - **do** $someComms()$
- **for** 1 to 10
  - **do** $someComms()$
- MPI_Call

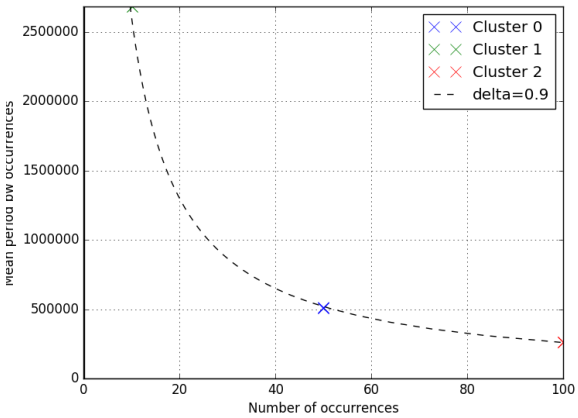


Figure 3.3: Clustering of delta example 2

In pseudocode 3.2.5 it can be seen the *isSubloop* implemented algorithm. What it does is considering the first mpi call of the target loop (b) as the boundaries of its iterations and same think for evaluated loop (a). Then the procedure is quite simple, it basically consists in counting how many iterations of a lies in one iteration of b. If any, then $a \mapsto b$. In the worst case, when a is not subloop of b the complexity is linear with number of iterations of b. Note that the implications defined in 3.3 are already fulfilled because the 3.2.3 algorithm. Number of iterations of a is greater than b because the sorting of loops and iteration mean time is lower because both belongs to the

same δ, it means that $it(a) * imt(a) = it(b) * imt(b)$ so $it(a) > it(b) \Rightarrow imt(a) < imt(b)$

Algorithm 3.2.5: IsSubloop function(a, b)

$itBounds \leftarrow T(firstCall(b))$
$subIts \leftarrow T(firstCall(a))$
if $subIts_0 < itBounds_0$
  then $itBounds \leftarrow \{0\} \cup itBouds$
for $i \in [0, |itBounds| - 1)$
  do
  $lowBound \leftarrow itBounds_i$
  $upBound \leftarrow itBounds_{i+1}$
  $inIts \leftarrow \forall x \in inIts : lowBound < x < upBound$
  if $|inIts| > 0$
    then return $(True)$
return $(False)$

At the end of this step we already have defined all relations between the different loops so the actual structure of the application have emerged. Internally every time a hierarchical relationship between two loops is found what the algorithm does is to move the loop from the δ set to literally inside the superloop, so at the end of the process on every $\delta \in \Delta$ what we have are the top level loops, defined by:

$$\forall a \in \delta, \nexists b \in \delta : a \mapsto b \tag{3.5}$$

3.2.4 Inter-rank reduction

Until now mpi calls with same call path have been considered different if they have been executed by different processes, in this step same call paths from different processes are merged, allowing to rank conditional execution to emerge. The first step is to perform the inter-rank reduction, and once done, the consecutive calls and loops with same processes are grouped into conditional blocks, easing the later step of pseudo-code construction.

In pseudocode 3.2.6 there is a description of how the inter-rank reduction is done. This function is called for every top level loops derived from the previous step. Assuming all items in the loop pased as parameter are ordered by call path, since same call path calls will be consecutive, the process of reducing is just about traverse the sequence of items and being collapsing same call path calls.

Remember that call paths consists on a sequence of pairs $(line, file)$. The only way to order them is by the value of line, so what we do when comparing two call paths is going from the top stack level to the bottom until there is a difference. Since it can be assumed two different branches are done from two different lines, the stack level after the last common level will be on same file,

so here is where the sort takes place taking the line value.

Algorithm 3.2.6: INTER RANK REDUCTION(*topLevelLoop*)

comment: Get set of mpi calls and subloops

$$
\begin{array}{l}
items \leftarrow items(topLevelLoop) \\
\textbf{for } i \in [0, |topLevelLoop|) \\
\quad \textbf{do} \begin{cases}
\textbf{if } isLoop(items_i) \\
\quad \textbf{then } InterRankReduction(items_i) \\
\quad \textbf{else} \begin{cases}
\textbf{for } j \in [i+1, |topLevelLoop|) \\
\quad \textbf{do} \begin{cases}
\textbf{if } isLoop(items_j) \\
\quad \textbf{then} \begin{cases} InterRankReduction(items_j) \\ break \end{cases} \\
\quad \textbf{else} \begin{cases}
\textbf{if } sameCallPath(item_i, item_j) \\
\quad \textbf{then} \begin{cases} reduce(item_i, item_j) \\ toremove(item_j) \end{cases} \\
\quad \textbf{else } break
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{array}
$$

The last step about group calls/loops into conditional blocks is a similar construction as before. It just travers all calls and group all consecutive calls/paths that are executed by the same set of processes. Every time a new call/loop presents to be executed by a different set of processes a new conditional block is built up.

Next step is about representing this information in such a way is easily understandable to the analyst.

3.2.5 Pseudo-code construction

The last step is about representation of the analysis. It is the interface with the user so is mandatory to provide a way to be as clear as possible. We want to show not just an static representation of the structure like in profilers but also loops structures that allow to understand the dynamic aspect of the execution. The structure also is enriched with information about its location on code performance information for the transitions between mpi calls that are indeed the actual computation. The alternative that we have found have been represent the application structure by means of a pseudocode. A toy example is depicted in figure 3.4.

In this example it can be seen that the output consists on 7 columns:

`FILE` and `LINE` Indicates the location from where the function on pseudocode in same line have been called.

`PSEUDOCODE` The actual representation of the structure of the application. Here you can see the loops and conditional structures when there is a divergence among ranks executions. Additionally it can be seen the call path taken for every mpi call. About mpi calls there is information about partner ranks in the parenthesis for p2p communications and the communicator id for collectives.

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 10 [id=2.0]	-	-	-
		: : FOR 1 TO 2.0 [id=0.0]	-	-	-
test-2.c	42	: : : CommSend()	-	-	-
		: : : : IF rank in [1]	-	-	-
test-2.c	14	: : : : : MPI_Send(1:0)	6.37us	4.0B	0.69
test-2.c	16	: : : : : MPI_Recv(1:1)	8.47us	4.0B	0.33
test-2.c	44	: : : CommRecv()	-	-	-
		: : : : IF rank in [0]	-	-	-
test-2.c	25	: : : : : MPI_Recv()	88.21us	-	0.7
test-2.c	26	: : : : : MPI_Send(0:1)	6.82us	4.0B	0.77
		: : END LOOP	-	-	-
		: : FOR 1 TO 5.0 [id=1.0]	-	-	-
test-2.c	49	: : : CommSend()	-	-	-
		: : : : IF rank in [1]	-	-	-
test-2.c	14	: : : : : MPI_Send(1:0)	6.08us	4.0B	0.82
test-2.c	16	: : : : : MPI_Recv(1:1)	8.54us	4.0B	0.3
test-2.c	51	: : : CommRecv()	-	-	-
		: : : : IF rank in [0]	-	-	-
test-2.c	25	: : : : : MPI_Recv()	168.88us	-	0.86
test-2.c	26	: : : : : MPI_Send(0:1)	6.27us	4.0B	0.79
		: : END LOOP	-	-	-
test-2.c	53	: : MPI_Barrier(CommId:1.0)	97.72us	-	0.44
		: END LOOP	-	-	-

Figure 3.4: Console GUI example

`E(time)` Is the mean time in every mpi call.

`E(size)` Is the mean size of the messages sended/recieved for every mpi call

`E(IPC)` Mean IPC both inside and outside mpi calls. In case of figure 3.4 there is not information about cpu burst in between mpi calls. You can see an example of how it looks like in figure 4.16.

Additionally an interactive shell have been developed, it allows the user to perform actions like show the cpu burst metrics, filter them by a given thresshold, filter information by mpi rank and show up the clustering plot

The process to transform the internal representation of the structure to the pseudocode is quite direct. First step is about removing repetitive information by means of functions `extractCommonCallstacks` and `hideContiguousCallstacks`. The former is about removing the common callstack shared by all calls/subloops belonging to the same conditional block or loop. Even if it can remove most of the repetitive information, there are cases where there is still information that is repeated but not common to the whole block, for this cases the second function is the responsible to remove it. It is better to understand with an example. In pseudocode 3.2.7 is the original output with all call path information for all mpi call. After extract the common callstack, we have the scenario depicted in 3.2.8 where there still are repeated information. Finally after remove contiguous call path we have 3.2.9.

Algorithm 3.2.7: ORIG.()

for 1 to N

do
$$\begin{cases} a:1 \rightarrow b:1 \rightarrow mpi \\ a:1 \rightarrow b:1 \rightarrow mpi \\ a:1 \rightarrow b:2 \rightarrow mpi \\ a:1 \rightarrow b:2 \rightarrow mpi \end{cases}$$

Algorithm 3.2.8: STEP 1()

$a : 1 \rightarrow$
for 1 to N
do $\begin{cases} b : 1 \rightarrow mpi \\ b : 1 \rightarrow mpi \\ b : 2 \rightarrow mpi \\ b : 2 \rightarrow mpi \end{cases}$

Algorithm 3.2.9: STEP 2()

$a : 1 \rightarrow$
for 1 to N
do $\begin{cases} b : 1 \\ \rightarrow mpi \\ \rightarrow mpi \\ b : 2 \\ \rightarrow mpi \\ \rightarrow mpi \end{cases}$

After remove repetitive information, next step is the parsing itself. We leave from having a set of phases Δ that consists on a set of top level loops, these loops consists on a set of mpi calls and other inner loops, so the point is to iterate over every delta, then over every loop and finally over every mpi call as is depicted in pseudocodes 3.2.10 and 3.2.11.

Algorithm 3.2.10: PSEUDOCODE CONSTRUCTION(Δ)

comment: Sort the top level loops by code position.

$\Delta \leftarrow sort(\Delta)$
for all $\delta \in \Delta$
do $\begin{cases} \textbf{for all } \upsilon \in \delta \\ \quad \textbf{do} \begin{cases} extractCommonCallstacks(\upsilon) \\ hideContiguousCallstacks(\upsilon) \end{cases} \end{cases}$

for all $\delta \in \Delta$
do $\begin{cases} \textbf{for all } \upsilon \in \delta \\ \quad \textbf{do } \{ parseLoopr(\upsilon) \end{cases}$

Algorithm 3.2.11: PARSELOOPR(*loop*)

comment: Sort calls and inner loops by code position

$loop \leftarrow sort(loop)$
for all $item \in loop$
do $\begin{cases} \textbf{if } item \in \Upsilon \\ \quad \textbf{then } parseLoopr(item) \\ \quad \textbf{else } togui(item) \end{cases}$

3.3 Further considerations

Until now we have been working from a simplistic point of view, in this section we are going to consider two main problems that have been arise that are specially harmful for the model described

above. The first one is the possibility to have aliasing. Can happen that two different loops are behaving in the same way in terms of the space described by iterations vs. iteration mean time, i.e. the model will detect mpi calls that belongs to two different loops are behaving equal so they will lie on the same cluster, considering them as part of the same loop. The second is that also can be possible that the same loop have mpi calls behaving in a different ways such that the model will detect more than one cluster of mpi calls that actually belongs to same loop. This behaviour can appear when the execution of an mpi call is conditioned by some data. For example imagine a situation where there is a loop of 10 iterations and a given mpi call is only executed on pair (or odd) iterations.

In order to overcome with these problems a data analysis on different loops features has been done looking for alternative or additional features to perform the clustering. This work is described in section 3.3.1. Since these analysis results did not end up with a better features set but just confirms our intuition enforcing the use of the current features set, some modifications of the basic methodology have been developed. They are described in section 3.3.2.

3.3.1 Loops feature selection

Additionally to the qualitative discussion done in section 3.1 about what features can unambiguously identify a loop, it is crucial to drive also a quantitative analysis. In order to do so, several loops iteration-level features have been collected from several executions and an analysis is then done over this data by means of two well-known methods [1]. Until now we were working just with mpi calls, in this section extra instrumentation of code is done in order to gather actual loops information, it is widely described in section 3.3.1.

The main objective for both techniques is to confirms or reject our intuition about number of iterations and iteration mean time, and to find out other possible features that can help on the task of identify loops by means of classification.

The first used technique is Principal Components analysis that allows to get hints about how features are related between them and with principal components. What we want to know with this technique is what loop features can better define them understanding as better features these subset of features that can explain the majority of the variability of the data being orthogonal between them, i.e. With poor correlation. To do so we present to the PCA algorithm a set of observations with several features every one, these observations are all the loops of the analyzed application.

The second technique is Variable Importance analysis using Random Forest method. This technique can determine what features are the most important for classification. In this case what we present to the algorithm is the same data that is outputted by the reduction phase, i.e. mpi calls but in this case with unambiguously information about to what loop every one of them belongs, i.e. to what cluster. It can be understood as a training data for supervised learning method.

[1] Feature selection is the process of selection a subset of relevant features for use in the model construction.

Data acquisition

Previously it has been discussed how mpi call number of repetitions and mean time between repetitions can be used for classify mpi calls into loops since they are a good proxy for number of iterations and iteration mean time so in order to assess this intuition these information will be gathered. Additionally i) IPC ii) and some instruction types counters will be also gathered. About the former, we would need mean number of instructions and mean number of cycles per iteration. The reason is that they maybe can contribute because some differences in the performance between loops. The intuition says that hardly can help for this purposes because, among other reasons, it depends a lot on other factors than the application. Remember in previous sections have been argued performance information is not optimal for syntactic structure analysis but we can not discard it categorically before do a quantitative analysis. The reason for the latest is that it can be thought if two loops performs different work, the relative quantity with total number of instructions of scalar, floating point or memory operations will differ. In order to do so, a set of hardware counters have been defined. The decision about what hardware counters pick have been dramatically restricted because the capabilities of the hardware used for the experiments, in this case MareNostrum 4. This set consist on: i) Number of instructions ii) number of cycles iii) number of unconditional branches iv) number of conditional branches v) total number of branches vi) and number of memory loads., additionally, number of floating point instructions for PCA analysis since traces in this case were taken from MinoTauro what provides this extra counter. Finally we also want to collect an unambiguously identification of loops since we want to be able to accurately relate these metrics to a given loop. The identification is done by hash function with code line and file.

As has been introduced above, trace does not have explicit information about loops so it should be collected from other sources. The reason is that the typical tracing method is by means of the `LD_PRELOAD` mechanism, so tracer library just can get information at shared library calls (like MPI, GOMP, ...). The alternative is to manually instrument the user code and fire events with the desired information by means of the tracer library API, in this case Extrae. Since the process of manually insert monitors presents to be tough for large codes has been decided to do it automatically using a source-to-source compiler, in this case Mercurium. Modifications on Mercurium have consisted on develop a new compilation phase that injects the desired monitors on the desired places in the code. Now, at execution time, those monitors fires the desired information to trace. All developments for this purpose are explained in annex B.1 in more detail.

Slightly different approaches have been applied in every case. In the case of the PCA, since we wanted to characterize loops, events are fired at the beginning and at the end of every loop marking the loop boundaries with just loop identification information, furthermore also events are fired at the beginning and at the end of loops iterations, in this case with all the mentioned hardware counters. Since tracefile presents to grow to dozens of Gigabytes for not so big executions and executions tends to be really stable just picking several iterations seems to be enough, so have been decided to instead of instrument all iterations of all loops, an iteration is instrumented given a probability. For these experiments the probability was set to 20%. Finally to keep track of how many iterations has been instrumented over the total, an extra event with total number of executed iterations (instrumented or not) is also fired to trace for every loop.

For random forest analysis data acquisition is done from another perspective, in this case what we wanted is the same information that will be available on production traces but labeled with the information of to what loop the mpi calls belongs to, so these traces can be understood as training traces. To do so the functions that marks entry/exit of loops does not fire any event to trace but keep track of the loops nesting hierarchy. Loop identification (or identifications when in a nested loops) is only fired to trace before an MPI call.

Data have been gathered for the follow NPB benchmarks with 128 ranks and input of class B: i) BT (121 ranks), ii) CG, iii) FT, iv) LU, v) MG, vi) SP IS and EP have not been included because the former just have one loop and so can not have aliasing problems and the second does not present repetitive patterns with MPI calls.

After data have been gathered to traces, post-process is done for prepare it for the performed analysis. On next section will be explained how this data have been manipulated and analyzed by means of the two selected methods.

Data analysis

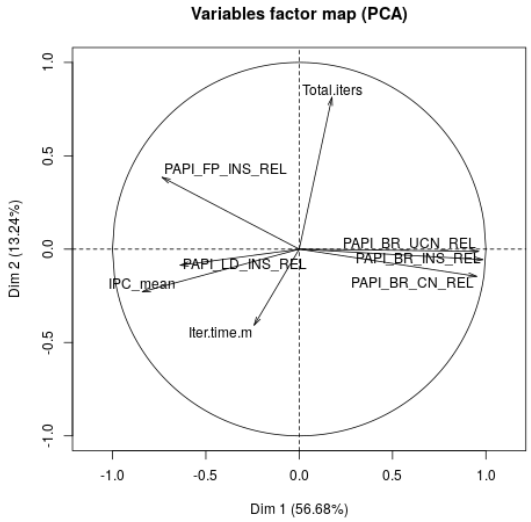
The first selected technique to drive the analysis has been the Principal Component Analysis (PCA): Data is linearly transformed in such a way it can be expressed in principal components that are sorted by the amount of variance that can explain and are orthogonal between them. It allows to identify patterns in data and expressing the data in such a way to highlight their similarities and differences. Once these patterns are found data can be compressed by reducing the number of dimensions with not much loss of information. To improve the understandability of the PCA, Variable factor map is used: It presents a view of the projection of the observed variables projected into the plane spanned by the first two principal components. This shows us the structural relationship between the variables and the components. The projection of a variable vector onto the component axis allows us to directly read the correlation between the variable and the component.

What we wanted to see through the PCA analysis is what features can better identify loops. To do so what is presented as input to the PCA procedure is a set of observations that are the loops with a set of features for every one of them. To prepare the data in this terms, a post-process of traces have been done. This post-process have consisted on aggregate all information from different iterations for every loop except for the special case of the number of iterations. In this last case total number of iterations is not the total number of instrumented iterations because not all of them have been instrumented but just a subset. Since the probability of an iteration to be instrumented is well-known (it has been set by us), the reconstructions is also trivial, being the total number of iterations described by equation 3.6.

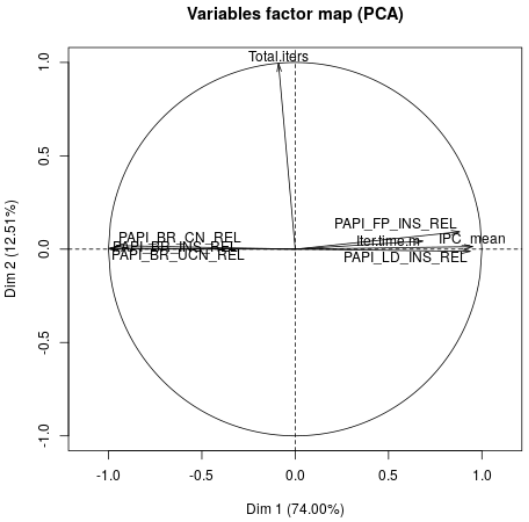
$$nit = iit * \frac{1}{\alpha^{\beta+1}} \tag{3.6}$$

Being α the probability to be instrumented and β the nesting level of the loop.

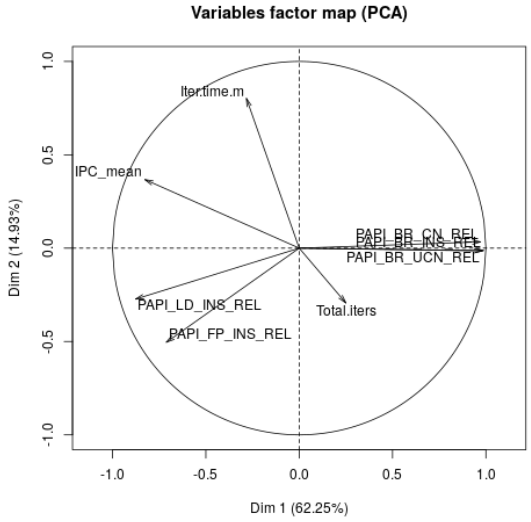
After apply PCA over the gathered data, on next figure the results can be seen. First principal component can explain the majority of the variance having a value of about 80% . It means that almost all data variance can be observed in the variable factor map.



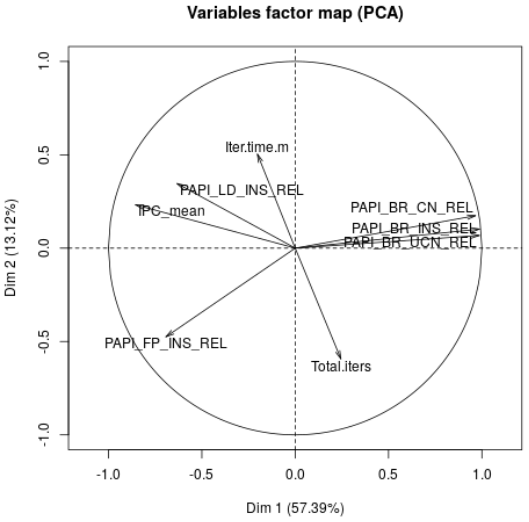
(a) BT



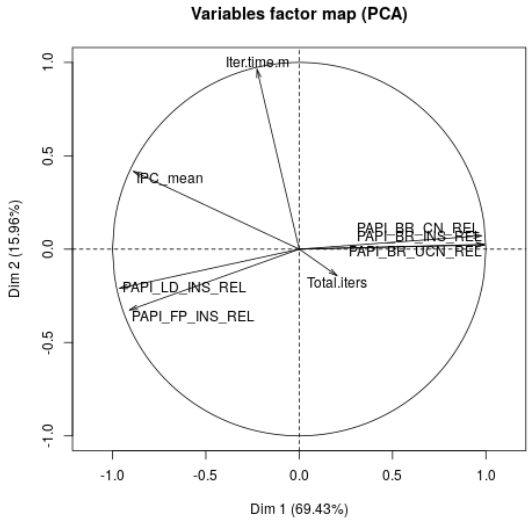
(b) CG



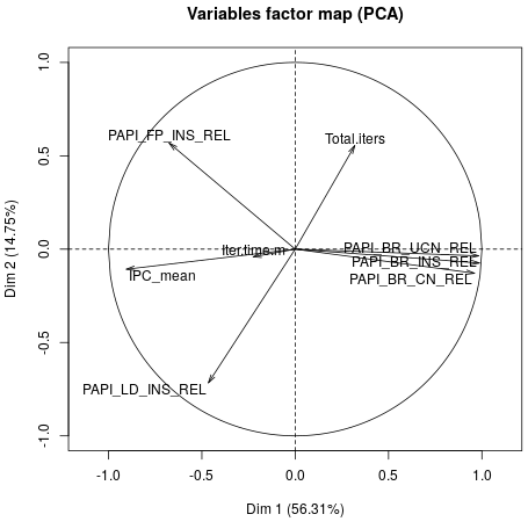
(c) FT



(d) LU



(e) MG



(f) SP

Figure 3.5: Variable factor map on NPB with both HWC merged

First thing to analyze is whether the first qualitative considerations done at the very beginning of this section are reinforced by this results or not. About total number of iterations it can be seen that is the variable better correlated to the second principal component in 3.5a, 3.5b, 3.5c and 3.5d. About mean iteration time it can be seen is highly correlated with first component in 3.5b and with the second in 3.5c and 3.5e. Both seems to be quite important in general so the first intuition was not bad at all. The bad news are that both metrics used to maintain a high correlation between them, in general negative, what is saying as more (dynamic) iteration a loop has, less iteration time. In this cases they are providing the same information, so one is not avoiding aliasing of the other. It have two lectures: i) It can be understood because the big loops that drives the execution are expensive because does a lot of computations and little function loops with simple jobs performs a lot of iterations like for example functions that looks for a character in a string. ii) The other lecture is that since we are counting dynamic iterations, subloops for sure will have more iterations that the big outer loops and also the iteration time for those big loops is inevitably bigger because they are containing those subloops. Nevertheless there are some situations where this correlation is not like in 3.5b where they are orthogonal. The conclusion is that these two metrics can explain a quite good amount of variability and can avoid aliasing but just in some cases.

Next thing to analyze is whether the IPC can help at the classification step, mostly on cases where number of iterations and iterations time are highly correlated that happens more obviously on 3.5a, 3.5c, 3.5d and 3.5e. In these cases it is used to present a moderated positive correlation with iteration mean time so better IPC when longest iterations. With this data the usefulness or not about IPC metric is fuzzy so more analysis needs to be done.

Lastly for PCA, it needs to check out how the instructions types counters are behaving. About branch instruction counters (PAPI_BR_UCN_REL, PAPI_BR_CN_REL and PAPI_BR_INS_REL) it can be said that in general they are strong positive correlated between them, presents a strong correlation with first component and is used to be orthogonal with iteration time mean and total number of iterations in cases when they are strongly correlated. Additionally since all three are explaining the same, picking just one should be enough. About the relative number of load instructions (PAPI_LD_INS_REL) it presents a quite different behavior depending on the execution but in most cases it presents a negative correlation with number of branches like in 3.5a, 3.5b, 3.5c and 3.5e what means that in most cases it can be used as well as branch counter when there is high correlation between total iterations and iterations mean time. Lastly, about relative floating point instructions (PAPI_FP_INS_REL) like before its behavior differs depending on the execution with respect to the rest of metrics but it is true that can explain quite a lot of variance.

The second used technique have been random forest for classification. It makes up a set of decision trees. Decision trees are trees that guides to a decision leaving from a set of features, in our case the decision is to what class a given item belongs, those trees are named classification trees. In these structures leaves represents class labels and branches conjunctions of features that lead to those class labels[2] [3]. The variable importance analysis is done over the classification trees and is expressed in terms of split purity, i.e. Whether the splits done over the data by using a

[2]https://en.wikipedia.org/wiki/Random_forest
[3]https://en.wikipedia.org/wiki/Decision_tree_learning

feature value drives to instances completely separated by classes or to a set of instances with mixed classes, being more pure as less mixture of classes. The purity is measured by the Gini index[4] being lower when purer split. So as more, one variable, contributes to decrease the Gini index, more importance it has.

The post-process in this case is basically the same as there is explained on section 3.2.1. Additionally, since we already have on trace to what loop every mpi call belongs to, this information is also gathered.

You can see the results in figure 3.6. First thing to discuss is whether the assumed important features on the previous qualitative discussion are also important on this quantitative analysis. The majority of the executions reinforce it since number of iterations and iteration mean time are the most important features for BT (3.6a), CG (3.6b), LU (3.6d) and for SP (3.6f). So it can be considered as a validation of the first intuition. In MG (3.6e) and FT (3.6c) cases these variables are specially irrelevant. About MG case the reason is because there are quite a lot of subloops doing 3 iterations. This pattern is used for communicate among the three dimensions of the model.

The results about the other features is not so encouraging since there is no a clear feature that have a good behaviour in general.

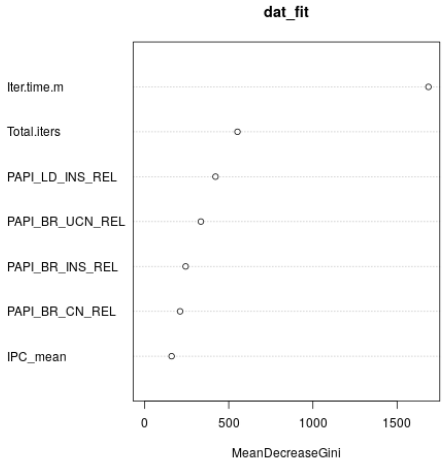
Summing up, clear improvements can not be achieved from these two analysis. PCA analysis have demonstrates how number of iterations and iteration mean time is good enough for some executions because can explain a lot of variability but not so good in other cases. Also there is no a clear sustitute or a clear additional feature to be used from this analysis. About variable importance analysis, in the same way, number of iterations a iterations mean time appears to be the most important variables and no alternatives can be easily selected from this analysis. More research have to be driven here.

3.3.2 Methodology modifications

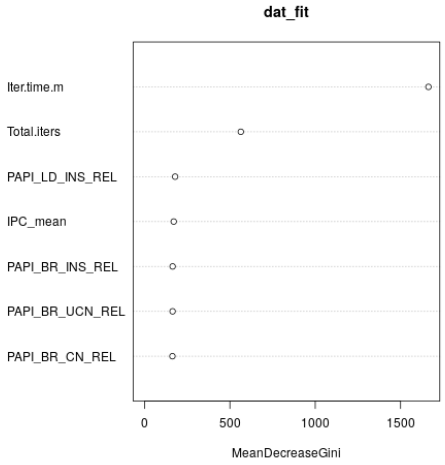
Ideally, cluster step must group mpi calls belonging to different loops into different clusters. For this purpose is critical to select the appropriate features and to do so we have driven data analysis explained in section 3.3. The results were not as good as we wanted, they have validate our first intuitions but can not provide extra or alternative features that can better drive the clustering. The features search have been left as a bottleneck to be solved in future works.

In order to deal with problematics that can not be solved from clustering, some measures have been taken. Such a harmful phenomena like aliasing or cluster splitting have drive to some modifications on the proposed tool. These modifications are basically extra checks that are capable of identify and solve these problems. In this section also is going to be explained a new problem that is not related with the clustering and therefore can not be overcome by improving it, we are talking about the “hidden superloop” problem.

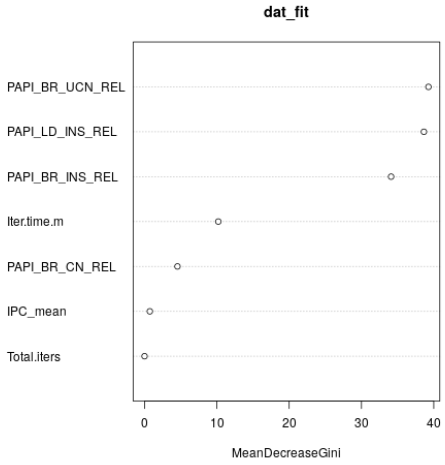
[4]Similarly the Gini index in economics is used for represent the wealth distribution in a nation being the most commonly used measurement for inequality.



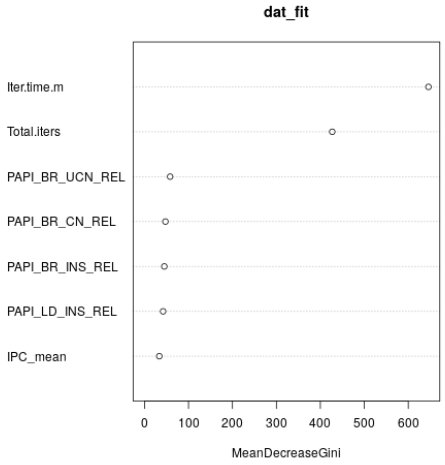
(a) BT



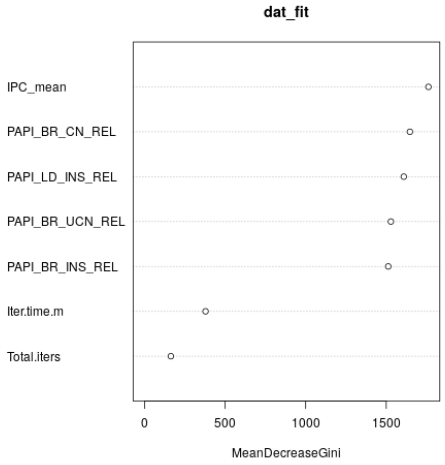
(b) CG



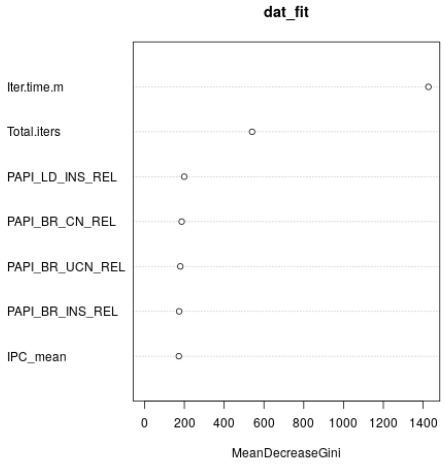
(c) FT



(d) LU



(e) MG



(f) SP

Figure 3.6: Variable importance by Random Forest method on NPB

Cluster Aliasing

When two loops are behaving in the same way in terms of number of iterations and iteration mean time there is no way to split them in the clustering step but something can be done by analyzing the timestamps of every mpi call.

The intuition is that when there are a set of mpi calls belonging to the same loop, their executions are interleaved, i.e. If a loop contains two mpi calls, lets say A, B its dynamic aspect will be $A, B, A, B \cdots$. If contrary they belongs to two different loops there is no interleaving any more and the dynamic aspect will be A, A, A, A, B, B, B, B. The way to detect it consist on three different phases:

i) Construct a matrix where rows are the different mpi calls in a cluster to being analyzed and cols are all its timestamps. Rows are sorted by call path, i.e. Mpi calls are sorted by program order and rows by timestamp value in ascending fashion.

ii) The matrix is transformed by adding voids on cells in such a way it fulfills that, traversing the matrix first by row and second by column, the next timestamp is always greater.

iii) If the matrix contains different loops, some voids have been added on previous point. This last step is about identify what we called "submatrixes" that are set of positions that are togheter with no voids between them, what indicates indeed the actual aliased loops in the cluster.

In can be better understood driving an example: Imagine we have the program depicted by pseudocode 3.3.1 the constructed matrix is depicted by table 3.1. For simplicity instead of timestamps we are using the order of execution, but the meaning is exactly the same. Then, following the second step, this matrix is transformed to what is depicted in table 3.2. The last step is about identify and retrieve the submatrixes. In this case there are two: {1,2,3,4}, {5,6,7,8}.

Algorithm 3.3.1: ALIASING EXAMPLE()

for 1 to 2
do $\begin{cases} MPI\_A() \\ MPI\_B() \end{cases}$
for 1 to 2
do $\begin{cases} MPI\_C() \\ MPI\_D() \end{cases}$

Cluster Splitting

On section 3.3.2 have been discussed the situation where two mpi calls that belongs to different loops lies on the same cluster, just the other way around in this section we are going to consider when two mpi calls belonging to the same loop, belongs to different clusters. With the used features for clustering it can happen for example when there is a data condition that makes one

Table 3.1: Matrix construction

MPI_A	1	3
MPI_B	2	4
MPI_C	5	7
MPI_D	6	8

Table 3.2: Matrix after step 2

MPI_A	1	3		
MPI_B	2	4		
MPI_C			5	7
MPI_D			6	8

mpi call to be executed in some iterations but no in others. An example can be seen in pseudocode 3.3.2. In this case the number of iterations reported by the MPI_A call will be 5 instead 10 also the iteration mean time will be doubled because there is a call to this mpi call every two loop iterations. What it means is that the MPI_A call will lie in different cluster but it still lies in same phase.

Algorithm 3.3.2: SPLIT EXAMPLE()

for $i = 1$ to 10

do $\begin{cases} \textbf{if } isPair(i) \\ \quad \textbf{then } MPI\_A() \\ MPI\_B() \\ MPI\_C() \end{cases}$

This situation has been solved in a very straightforward way. In section 3.2.3 have been explained how loops are merged from most iterations loops to few iterations because subloops will for sure have more iterations than the superloops. In the scenario we are already discussing this is not true anymore becase data conditions. The implemented solution have been first try to merge loops in a given phase from few to most iterations loops, the so called "reverse loop mergin" performing a similar times checking as in *isSubloop* function depicted in 3.2.5. Once it is done, the actual loops merging is performed.

Hidden superloop detection

The way we are detecting loops is by means of mpi calls, what it implies is that there is no clue about those loops that does not have any call to the mpi library. In general we do not care about them but there are some cases where they are important, like is in the case when there are undetectable superloops that contains detectable subloops. Imagine the situation on the pseucode 3.3.3, there is a undetectable superloop with 10 iterations that contains a detectable loop of 2 iterations. If we apply the methodology described in this chapter we will end up with a

Table 3.3: Matrix construction

MPI_A	1	3	9	11
MPI_B	2	4	10	12
MPI_C	5	7	13	15
MPI_D	6	8	14	16

unique loop of 20 iterations. This derived loop is correct since if we replay the 20 iterations loop, its dynamic aspect is the same like the nested loop. Going a bit further, now imagine we have a program with the aspect depicted in pseudo-code 3.3.4. Now our model will end up with two loops with four iterations every one. That result is not representing what is actually the structure of the application, so it is a wrong output.

Algorithm 3.3.3: ALIASING EXAMPLE()

for 1 to 10
do { **for** 1 to 2
do { $MPI\_A()$
$MPI\_B()$

The solution is just add one more step to the solution presented for clustering aliasing in 3.3.2. Now if we analyze the timestamps of mpi calls in example 3.3.4 we will obtain matrix 3.3 and then apply the three steps we will end up with the matrix in table 3.4. Since same mpi call can not belongs to more than one loop, then the split is done on the vertical axis. In this case two subloops have been detected. The additional step is about counting how many submatrixes every detected subloop have, this number is the number of the iterations of the "hidden superloop".

Algorithm 3.3.4: ALIASING EXAMPLE()

for 1 to 2
do { **for** 1 to 2
do { $MPI\_A()$
$MPI\_B()$
for 1 to 2
do { $MPI\_C()$
$MPI\_D()$

Table 3.4: Matrix after step 2

MPI_A	1	3			9	11		
MPI_B	2	4			10	12		
MPI_C			5	7			13	15
MPI_D			6	8			14	16

Chapter 4

Results

All the ideas exposed on previous chapter have been put into practice in order to assess the quality of the results. The resulting piece of software does not intend to be pretty polished and well-designed software, neither to be very refined in terms of performance but just to be a prototype. With this idea in mind the used language have been Python because its easy-of-use is a good choice for rapid prototyping software.

In this chapter the quality of results are assessed and the scalability of the tool is analyzed.

4.1 Validation

Validation consists on check out if the actual results happen to meet with what was expected. In our case we expect the resulting pseudocode can represent the general structure of the target application. The strategy followed for validation will be starting with some toy examples in order to validate concrete cases and then perform a validation in a real world application.

4.1.1 Specific capabilities validation

First step of validation is to check out the behavior of the prototype for the expected scenarios. The strategy is to start with toy examples and try to increase the difficulty by adding complexity.

Nesting loops

In this test is going to be checked if subloops at different nesting levels are well detected. The body of the most inner loop just consists on two p2p operations, the pair ranks performs a recv while odd ranks a send. For the loops of the rest of the levels also contains an mpi call, because if not they will be completely invisible for us.

One single loop detection is trivial so lets start the validation with 2 and 3 nested loops. In figure 4.1 you can see the clustering performed for the two nested loops example, meanwhile in figure 4.2 there is the clustering for the three levels of nested loops. In these figures it can be seen how there is the same number of clusters as loops and in both cases all of them belonging to the same phase, i.e. Explains the same amount of time of the application, what in our model indicates

that could be a hierarchical relationship between them that will be discovered on the loops merge step. Looking on this plots, the loops merge step is done in a bottom-right to top-left fashion.

After execute the rest of steps, the output of the first case is depicted in figure 4.3. The second case output is in figure 4.4.

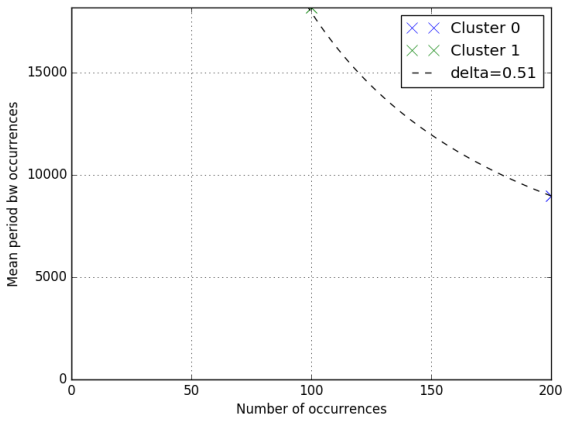


Figure 4.1: Clustering of validation example 1

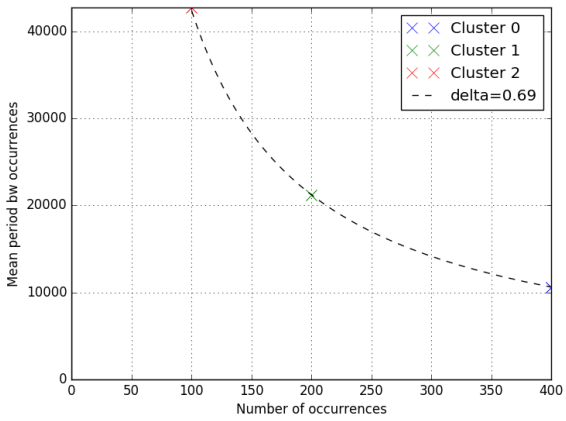


Figure 4.2: Clustering of validation example 2

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 100 [id=1.0]	-	-	-
test-2.c	17	: : MPI_Barrier(CommId:1.0)	5.61us	-	1.07
		: : FOR 1 TO 2.0 [id=0.0]	-	-	-
		: : : IF rank in [1]	-	-	-
test-2.c	23	: : : : MPI_Send(1:0)	4.84us	4.0B	1.16
		: : : IF rank in [0]	-	-	-
test-2.c	25	: : : : MPI_Recv()	5.13us	0.02B	1.17
		: : END LOOP	-	-	-
		: END LOOP	-	-	-

Figure 4.3: Result for 2 nested loops

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 100 [id=1.0]	-	-	-
test-3.c	17	: : MPI_Barrier(CommId:1.0)	5.74us	-	1.03
		: : FOR 1 TO 2.0 [id=0.0]	-	-	-
test-3.c	20	: : : MPI_Barrier(CommId:1.0)	5.35us	-	1.14
		: : : FOR 1 TO 2.0 [id=2.0]	-	-	-
		: : : : IF rank in [1]	-	-	-
test-3.c	26	: : : : : MPI_Send(1:0)	4.83us	4.0B	1.15
		: : : : IF rank in [0]	-	-	-
test-3.c	28	: : : : : MPI_Recv()	5.18us	0.01B	1.22
		: : : END LOOP	-	-	-
		: : END LOOP	-	-	-
		: END LOOP	-	-	-

Figure 4.4: Result for 3 nested loops

It has been demonstrated that the prototype is capable to detect an arbitrary number of nested loops (just 3 levels have been shown here for space reasons).

Phases detection

Have been claimed that the proposed model can detect different phases of the execution, i.e. Different loops with no hierarchical relationship that explains different parts of the execution. In this section there is going to be validated if it can be done in an scenario with arbitrary number of different phases having an arbitrary nested level on every one of them. For space reasons the experiment done is just for 3 phases, explaining different amount of time, with two nested levels on every phase.

In figure 4.5 there is the clustering for the exposed example and in figure 4.6 the outputted pseudo-code. In the clustering can be clearly seen the three different phases explaining 11%, 25% and 42% of the overall execution time respectively. It fits very well with we can see on pseudocode. Taking into account that in this example every iteration takes the same amount of time (with a certain variability), the first loop executes $\frac{50}{350} \approx 15\%$ of the iterations, second loop $\frac{200}{350} \approx 57\%$ and the third $\frac{100}{350} \approx 29\%$. Those values are strongly related with the deltas seen in the clustering. Additionally for every phase it can be seen two different clusters that are merged in the same way as explained for the "nesting loops" example.

With this example the multiple phases detection is demonstrated to work correctly also together with multiple nesting loops.

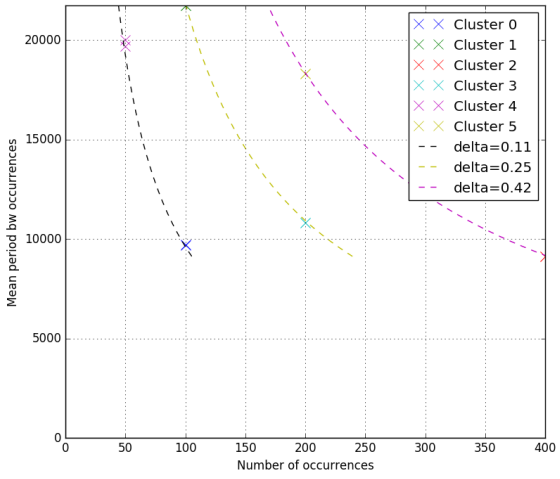


Figure 4.5: Clustering of validation example 3

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 50 [id=4.0]	-	-	-
test-4.c	17	: : MPI_Barrier(CommId:1.0)	6.48us	-	0.96
		: : FOR 1 TO 2.0 [id=0.0]	-	-	-
		: : : IF rank in [1]	-	-	-
test-4.c	23	: : : MPI_Send(1:0)	5.03us	4.0B	1.07
		: : : IF rank in [0]	-	-	-
test-4.c	25	: : : MPI_Recv()	5.56us	-	1.02
		: : END LOOP	-	-	-
		: END LOOP	-	-	-
	0	main()	-	-	-
		: FOR 1 TO 200 [id=5.0]	-	-	-
test-4.c	31	: : MPI_Barrier(CommId:1.0)	5.55us	-	1.09
		: : FOR 1 TO 2.0 [id=2.0]	-	-	-
		: : : IF rank in [1]	-	-	-
test-4.c	37	: : : MPI_Send(1:0)	4.88us	4.0B	1.16
		: : : IF rank in [0]	-	-	-
test-4.c	39	: : : MPI_Recv()	5.16us	0.02B	1.17
		: : END LOOP	-	-	-
		: END LOOP	-	-	-
	0	main()	-	-	-
		: FOR 1 TO 100 [id=1.0]	-	-	-
test-4.c	45	: : MPI_Barrier(CommId:1.0)	6.53us	-	0.93
		: : FOR 1 TO 2.0 [id=3.0]	-	-	-
		: : : IF rank in [1]	-	-	-
test-4.c	51	: : : MPI_Send(1:0)	5.83us	4.0B	0.94
		: : : IF rank in [0]	-	-	-
test-4.c	53	: : : MPI_Recv()	6.24us	-	1.16
		: : END LOOP	-	-	-
		: END LOOP	-	-	-

Figure 4.6: Result for nested loops on different phases

Nested loops with aliasing

The two previous tests were without taking into account the aliasing that could appear in some cases so in order to assess the modifications introduced in section 3.3.2 works well, some tests have been prepared. This test consists on assess the detection of arbitrary nested aliased loops. So there is a superloop with mpi calls, in order to detect it, with some aliased subloops in its body. For space reasons, two examples are done with two and three nested aliased loops respectively.

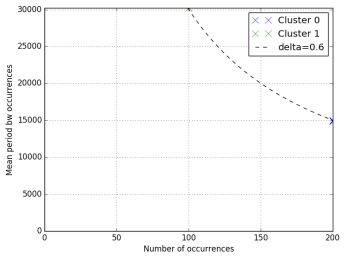


Figure 4.7: Clustering for 2 nested aliased loops

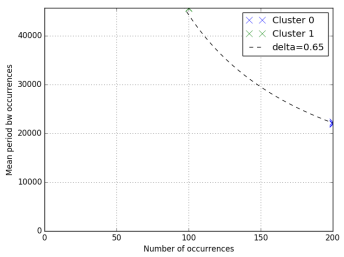


Figure 4.8: Clustering of 3 nested aliased loops

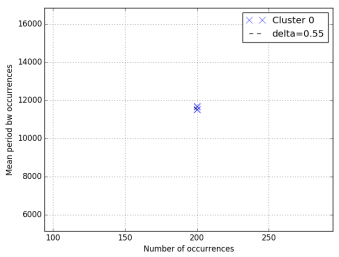


Figure 4.9: Clustering of 3 nested aliased loops with hidden superloop

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 100 [id=1.0]	-	-	-
test-5.c	17	: : MPI_Barrier(CommId:1.0)	6.05us	-	1.06
		: : FOR 1 TO 2.0 [id=0.0]	-	-	-
		: : : IF rank in [1]	-	-	-
test-5.c	23	: : : : MPI_Send(1:0)	4.87us	4.0B	1.14
		: : : IF rank in [0]	-	-	-
test-5.c	25	: : : : MPI_Recv()	5.19us	-	1.14
		: : END LOOP	-	-	-
		: : FOR 1 TO 2.0 [id=0.1]	-	-	-
		: : : IF rank in [1]	-	-	-
test-5.c	32	: : : : MPI_Send(1:0)	4.77us	4.0B	1.21
		: : : IF rank in [0]	-	-	-
test-5.c	34	: : : : MPI_Recv(0:0)	5.03us	0.12B	1.15
		: : END LOOP	-	-	-
		: END LOOP	-	-	-

Figure 4.10: Result for 2 nested aliased loops

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 100 [id=1.0]	-	-	-
test-6.c	17	: : MPI_Barrier(CommId:1.0)	7.05us	-	0.86
		: : FOR 1 TO 2.0 [id=0.0]	-	-	-
		: : : IF rank in [1]	-	-	-
test-6.c	23	: : : : MPI_Send(1:0)	5.33us	4.0B	0.97
		: : : IF rank in [0]	-	-	-
test-6.c	25	: : : : MPI_Recv()	5.7us	-	0.99
		: : END LOOP	-	-	-
		: : FOR 1 TO 2.0 [id=0.1]	-	-	-
		: : : IF rank in [1]	-	-	-
test-6.c	32	: : : : MPI_Send(1:0)	5.08us	4.0B	1.02
		: : : IF rank in [0]	-	-	-
test-6.c	34	: : : : MPI_Recv()	5.37us	-	1.09
		: : END LOOP	-	-	-
		: : FOR 1 TO 2.0 [id=0.2]	-	-	-
		: : : IF rank in [1]	-	-	-
test-6.c	41	: : : : MPI_Send(1:0)	5.11us	4.0B	1.05
		: : : IF rank in [0]	-	-	-
test-6.c	43	: : : : MPI_Recv()	5.34us	-	1.07
		: : END LOOP	-	-	-
		: END LOOP	-	-	-

Figure 4.11: Result for 3 nested aliased loops

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 100 [id=0.-1]	-	-	-
		: : FOR 1 TO 2.0 [id=0.0]	-	-	-
		: : : IF rank in [1]	-	-	-
test-7.c	22	: : : : MPI_Send(1:0)	4.96us	4.0B	1.12
		: : : IF rank in [0]	-	-	-
test-7.c	24	: : : : MPI_Recv()	5.07us	2.42B	1.15
		: : END LOOP	-	-	-
		: : FOR 1 TO 2.0 [id=0.1]	-	-	-
		: : : IF rank in [1]	-	-	-
test-7.c	31	: : : : MPI_Send(1:0)	4.89us	4.0B	1.18
		: : : IF rank in [0]	-	-	-
test-7.c	33	: : : : MPI_Recv()	4.96us	2.38B	1.27
		: : END LOOP	-	-	-
		: END LOOP	-	-	-

Figure 4.12: Result for 2 nested aliased loops with hidden superloop

If you look at the clustering you will see how just two clusters appears in both cases. The top-left cluster is representing the outer loop meanwhile the bottom-right represents both inner loops (figure 4.7) in first example and all three for the second (figure 4.8), here is where the aliasing appears. For detecting the multiple aliased loops the technique described in 3.3.2 is applied and the results are satisfactory as you can see on outputted pseudo-codes 4.10 and 4.11.

Additionally has been tested the "hidden superloop" detection just by removing the `MPI_Barrier` in line 17. It will cause the top-left cluster representing the outer loop disappear as you can see on

clustering in figure 4.9 and the results on figure 4.12. This superloop have been detected because the additional modifications introduced in section 3.3.2.

Data conditions

Last thing to check out is whether the model can detect loops with some mpi calls under data conditions. This scenario was introduced in section 3.3.2 where also the modifications introduced to the model are explained. Could happen that some communications are executed just for some set of iterations, for example in order to save communications overhead. In this last test two examples are done.

First of them will show how in some situations even if there an mpi call under a data condition, the model can solve it without the extra checks introduced in 3.3.2 building up a valid representation of the applications' structure. It will consist on a mpi call at the beginning of the loop body that is executed just in odd iterations, then after that a typical send for odd ranks and recv for pair ranks is executed. The results can be seen in figure 4.13. Even if the original code was one single loop of 100 iterations with the `MPI_Barrier` being executed just for odd iterations, the representation outputted by the model is one outer loop of 50 iterations and one inner loop of 2 iterations. This is completely valid representation that can be seen if we compare the dynamic aspect of both the original and the outputted loop.

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 50 [id=1.0]	-	-	-
test-8.c	18	: : MPI_Barrier(CommId:1.0)	5.55us	-	0.98
		: : FOR 1 TO 2.0 [id=0.0]	-	-	-
		: : : IF rank in [1]	-	-	-
test-8.c	23	: : : : MPI_Send(1:0)	5.12us	4.0B	1.16
		: : : IF rank in [0]	-	-	-
test-8.c	25	: : : : MPI_Recv()	5.41us	-	1.13
		: : END LOOP	-	-	-
		: END LOOP	-	-	-

Figure 4.13: Result for mpi call under data condition 1

Second example is about when the extra check mechanism must act. It shares about the same structure of the first example but with additional communications executed before the conditioned mpi call. The key point is that now the conditioned call is surrounded by non conditioned calls. You can see the result for this second example on figure 4.14. Since we do not have information about to with what data the execution of a given mpi call is conditioned, the way to depict a given call is being executed under a data condition is by means of what it can be seen in line 24 of the result. It represents the probability to be executed. Have the knowledge about in what iterations are these kind of mpi calls executed could be a big deal so future developments in this sense have to be done.

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 100	-	-	-
		: : IF rank in [1]	-	-	-
test-9.c	20	: : : MPI_Send(1:0)	5.01us	4.0B	1.1
		: : IF rank in [0]	-	-	-
test-9.c	22	: : : MPI_Recv()	5.3us	-	1.03
test-9.c	24	: : 50.0% => MPI_Barrier(CommId:1)	5.99us	-	0.97
		: : IF rank in [1]	-	-	-
test-9.c	26	: : : MPI_Send(1:0)	4.9us	4.0B	1.14
		: : IF rank in [0]	-	-	-
test-9.c	28	: : : MPI_Recv()	5.23us	-	1.1
		: END LOOP	-	-	-

Figure 4.14: Result for mpi call under data condition 2

4.1.2 Real application validation

The proposal explained in this thesis is devoted to deal with real world applications so the last step of validation is to assess the correctness of the analysis for them. For time reasons just one application has been analyzed for medium size execution.

There are two ways to assess the quality of the results in this case:

i) Inspecting manually the source code of the application looking for the general structure.

ii) Using Paraver visualizer. With paraver it can be analyzed where and when every mpi call has taken place and the order of them.

The first presents to be very tough and error prone when dealing with huge source codes so it is discarded. Using paraver is a good alternative since the general structure can be easily visualized and for sure the performance information.

Lulesh 2.0

Lulesh is understood as "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics". It approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh[1].

The execution have been done with 128 mpi ranks and for 30 iterations. On figure 4.15 you can see the clustering. It seems a quite simple application since just appears one cluster. It would mean that there is just one loop with no nesting loops. In figure 4.16 there is the outputted pseudocode. For space reasons, this code have been previously filter in order to only show the structure for rank 0. Since Lulesh is a SPMD program we can expect the pseudo-code for all the 128 ranks to be quite similar so only the presented one will be validated. Additionally the threshold on computation phases have been set on 6ms.

First thing to take into account is that this is a single loop with no nested loops that executes 30 iterations. This first part is quite easy to check out looking to the trace that corresponds to this execution on figure 4.17. It can be easily identified an structure that is being repeated during the whole execution. Even if it is a bit tricky it can be counted that there are 30 repetitions. If now we enter to the body of the loop the first think it can be seen is that there is an `MPI_Allreduce` on the communicator 1 that is executed the 96,6% of the times that corresponds to 29 iterations what

[1] https://codesign.llnl.gov/lulesh.php

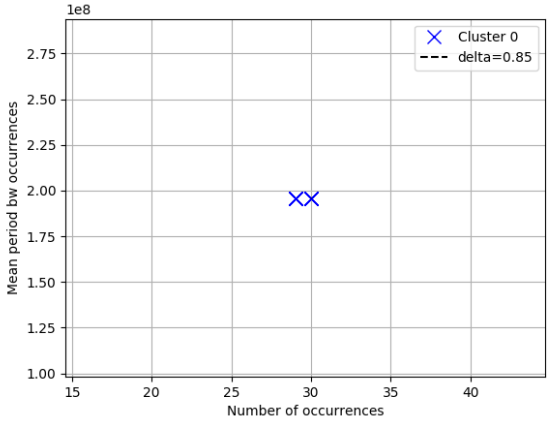


Figure 4.15: Clustering of lulesh 2.0 128 ranks 30 iterations

is validated in figure 4.18 (this table contains the mpi call counts for the body of the execution, cutting initialization and finalization). About the rest of the mpi calls also can be validated with the same table by counting the times that appears on the pseudo-code and multiply it by the number of iterations. It is demonstrated how the mpi calls count is related perfectly with what can be seen on trace.

i) `MPI_Comm_rank` $9 * 30 = 270$

ii) `MPI_Comm_rank` $17 * 30 = 510$

iii) `MPI_Comm_rank` $10 * 30 = 300$

iv) `MPI_Comm_rank` $3 * 30 = 90$

v) `MPI_Comm_rank` $17 * 30 = 510$

We already have seen how, at least, the number of times every mpi call is executed corresponds with the actual execution, now we need to figure out if the presented structure of the body of the loop is correct. To do so we can compare the different big computational phases that appears both in pseudocode and in trace. For this purpose in figure 4.19 it can be seen a detailed view for a given iteration. The times for the different phases have been taken from the rank 0 even if in trace there are shown all ranks. This is because by this way is much easier to visualize the structure of the iteration. There are five long computational phases and if we perform a comparison between the computational phases detected by our tool and the phases on trace we can realize they are the same and in the same order so have been demonstrated the structure detected is correct. We can perform the same comparison for the IPC with same result. Additionally for assess the mpi calls are correctly arranged between these computational phases, the figure 4.20 can be consulted. Here the mpi call count have been divided into 5 parts that corresponds to the splits that the different big computational phases does.

By analyzing the trace have been demonstrated how the developed tool can satisfactorily detect the internal structure of a well-known real application, in this case, Lulesh.

FILE	LINE	PSEUDOCODE	E(TIME)	E(SIZE)	E(IPC)
	0	main()	-	-	-
		: FOR 1 TO 30 [id=0.0]	-	-	-
lulesh.cc	2773	: : TimeIncrement()	-	-	-
lulesh.cc	**214**	**: : : [~ computation ~]**	**56.63ms**	**-**	**2.56**
lulesh.cc	214	: : : 96.6% => MPI_Allreduce(CommId:1)	35.32us	8.0B/8.0B	1.57
lulesh.cc	2774	: : LagrangeLeapFrog()	-	-	-
lulesh.cc	2648	: : : LagrangeNodal()	-	-	-
lulesh.cc	1263	: : : : CalcForceForNodes()	-	-	-
lulesh.cc	1137	: : : : : CommRecv()	-	-	-
lulesh-comm.cc	101	: : : : : : MPI_Comm_rank(0:)	10.27us	-	0.28
lulesh-comm.cc	119	: : : : : : MPI_Irecv(0:25)	10.51us	22.52KB	0.55
lulesh-comm.cc	137	: : : : : : MPI_Irecv(0:5)	8.67us	22.52KB	0.96
lulesh-comm.cc	155	: : : : : : MPI_Irecv(0:1)	8.43us	22.52KB	1.08
lulesh-comm.cc	192	: : : : : : MPI_Irecv(0:6)	8.8us	744.0B	1.08
lulesh-comm.cc	201	: : : : : : MPI_Irecv(0:30)	8.54us	744.0B	1.25
lulesh-comm.cc	210	: : : : : : MPI_Irecv(0:26)	8.29us	744.0B	1.25
lulesh-comm.cc	345	: : : : : : MPI_Irecv(0:31)	8.32us	24.0B	0.9
lulesh.cc	1158	: : : : : CommSend()	-	-	-
lulesh-comm.cc	**401**	**: : : : : : [~ computation ~]**	**85.2ms**	**-**	**2.48**
lulesh-comm.cc	401	: : : : : : MPI_Comm_rank(0:)	20.08us	-	1.59
lulesh-comm.cc	436	: : : : : : MPI_Isend(0:25)	14.95us	22.52KB	1.9
lulesh-comm.cc	477	: : : : : : MPI_Isend(0:5)	10.07us	22.52KB	1.65
lulesh-comm.cc	518	: : : : : : MPI_Isend(0:1)	10.5us	22.52KB	0.93
lulesh-comm.cc	589	: : : : : : MPI_Isend(0:6)	14.79us	744.0B	1.68
lulesh-comm.cc	606	: : : : : : MPI_Isend(0:30)	12.45us	744.0B	1.86
lulesh-comm.cc	623	: : : : : : MPI_Isend(0:26)	13.13us	744.0B	1.66
lulesh-comm.cc	837	: : : : : : MPI_Isend(0:31)	9.48us	24.0B	0.84
lulesh-comm.cc	843	: : : : : : MPI_Waitall(0:)	541.3us	-	0.43
lulesh.cc	1161	: : : : : CommSBN()	-	-	-
lulesh-comm.cc	889	: : : : : : MPI_Comm_rank(0:)	9.32us	-	0.49
lulesh-comm.cc	911	: : : : : : MPI_Wait(0:)	9.25us	-	0.6
lulesh-comm.cc	945	: : : : : : MPI_Wait(0:)	8.6us	-	2.48
lulesh-comm.cc	980	: : : : : : MPI_Wait(0:)	8.52us	-	2.35
lulesh-comm.cc	1039	: : : : : : MPI_Wait(0:)	8.41us	-	1.72
lulesh-comm.cc	1053	: : : : : : MPI_Wait(0:)	8.19us	-	1.64
lulesh-comm.cc	1067	: : : : : : MPI_Wait(0:)	8.31us	-	1.98
lulesh-comm.cc	1251	: : : : : : MPI_Wait(0:)	58.99us	-	1.44
lulesh.cc	1267	: : : : CommRecv()	-	-	-
lulesh-comm.cc	101	: : : : : MPI_Comm_rank(0:)	7.77us	-	0.73
lulesh-comm.cc	119	: : : : : MPI_Irecv(0:25)	9.03us	45.05KB	0.79
lulesh-comm.cc	137	: : : : : MPI_Irecv(0:5)	7.37us	45.05KB	1.02
lulesh-comm.cc	155	: : : : : MPI_Irecv(0:1)	7.34us	45.05KB	0.95
lulesh-comm.cc	192	: : : : : MPI_Irecv(0:6)	7.6us	1.45KB	1.12
lulesh-comm.cc	201	: : : : : MPI_Irecv(0:30)	7.51us	1.45KB	1.33
lulesh-comm.cc	210	: : : : : MPI_Irecv(0:26)	7.42us	1.45KB	1.13
lulesh-comm.cc	345	: : : : : MPI_Irecv(0:31)	7.5us	48.0B	0.95
lulesh.cc	1289	: : : : CommSend()	-	-	-
lulesh-comm.cc	**401**	**: : : : : [~ computation ~]**	**6.15ms**	**-**	**2.0**
lulesh-comm.cc	401	: : : : : MPI_Comm_rank(0:)	14.58us	-	1.63
lulesh-comm.cc	843	: : : : : MPI_Waitall(0:)	10.26us	-	0.35
lulesh.cc	1292	: : : : CommSyncPosVel()	-	-	-
lulesh-comm.cc	1310	: : : : : MPI_Comm_rank(0:)	22.14us	-	0.73
lulesh-comm.cc	1332	: : : : : MPI_Wait(0:)	274.73us	-	0.79
lulesh-comm.cc	1366	: : : : : MPI_Wait(0:)	72.95us	-	2.46
lulesh-comm.cc	1402	: : : : : MPI_Wait(0:)	23.24us	-	2.3
lulesh-comm.cc	1461	: : : : : MPI_Wait(0:)	198.36us	-	1.79
lulesh-comm.cc	1475	: : : : : MPI_Wait(0:)	7.83us	-	1.83
lulesh-comm.cc	1489	: : : : : MPI_Wait(0:)	7.75us	-	2.08
lulesh-comm.cc	1674	: : : : : MPI_Wait(0:)	333.07us	-	1.73
lulesh.cc	2656	: : : LagrangeElements()	-	-	-
lulesh.cc	2461	: : : : CalcQForElems()	-	-	-
lulesh.cc	1992	: : : : : CommRecv()	-	-	-
lulesh-comm.cc	**101**	**: : : : : : [~ computation ~]**	**29.32ms**	**-**	**2.25**
lulesh-comm.cc	101	: : : : : : MPI_Comm_rank(0:)	18.76us	-	1.39
lulesh-comm.cc	119	: : : : : : MPI_Irecv(0:25)	12.61us	21.09KB	0.43
lulesh-comm.cc	137	: : : : : : MPI_Irecv(0:5)	9.23us	21.09KB	1.01
lulesh-comm.cc	155	: : : : : : MPI_Irecv(0:1)	8.79us	21.09KB	1.17
lulesh.cc	2010	: : : : : CommSend()	-	-	-
lulesh-comm.cc	**401**	**: : : : : : [~ computation ~]**	**15.35ms**	**-**	**1.91**
lulesh-comm.cc	401	: : : : : : MPI_Comm_rank(0:)	15.24us	-	1.52
lulesh-comm.cc	436	: : : : : : MPI_Isend(0:25)	13.14us	21.09KB	1.9
lulesh-comm.cc	477	: : : : : : MPI_Isend(0:5)	10.28us	21.09KB	1.89
lulesh-comm.cc	518	: : : : : : MPI_Isend(0:1)	10.57us	21.09KB	1.26
lulesh-comm.cc	843	: : : : : : MPI_Waitall(0:)	291.68us	-	0.45
lulesh.cc	2014	: : : : : CommMonoQ()	-	-	-
lulesh-comm.cc	1734	: : : : : : MPI_Comm_rank(0:)	9.51us	-	0.45
lulesh-comm.cc	1757	: : : : : : MPI_Wait(0:)	8.88us	-	0.81
lulesh-comm.cc	1791	: : : : : : MPI_Wait(0:)	8.78us	-	2.04
lulesh-comm.cc	1824	: : : : : : MPI_Wait(0:)	8.53us	-	2.13
		: END LOOP	-	-	-

Figure 4.16: Intern structure of lulesh 2.0 - Rank 0 - Computation threshold 6ms

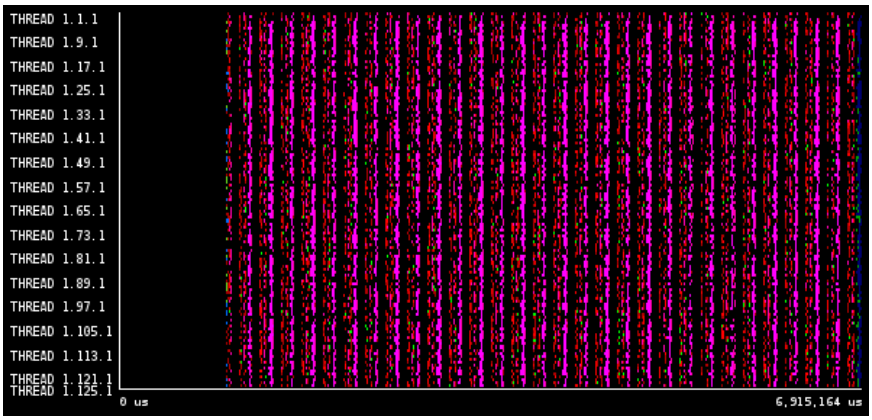


Figure 4.17: Lulesh2.0 128 mpi ranks - 30 iterations trace

	MPI_Isend	MPI_Irecv	MPI_Wait	MPI_Waitall	MPI_Allreduce	MPI_Comm_rank
THREAD 1.1.1	300	510	510	90	29	270

Figure 4.18: Lules2.0 mpi call count for rank 0

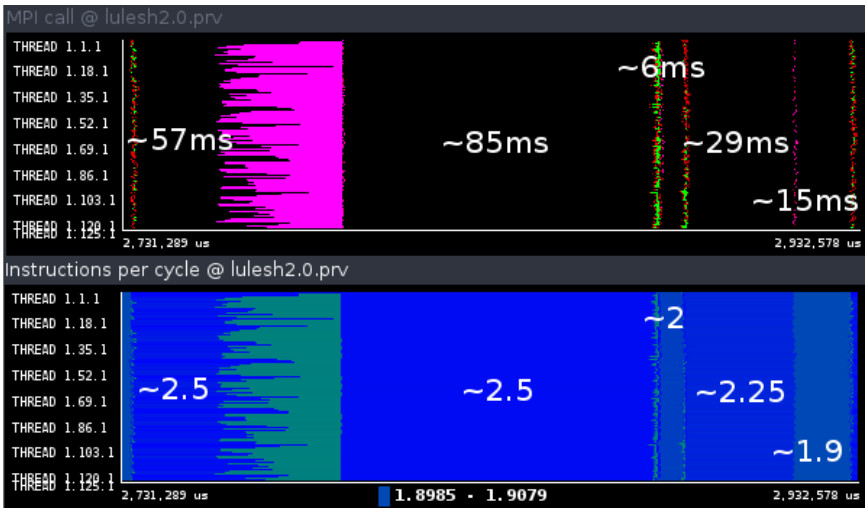


Figure 4.19: Lules2.0 detailed view for a single iteration

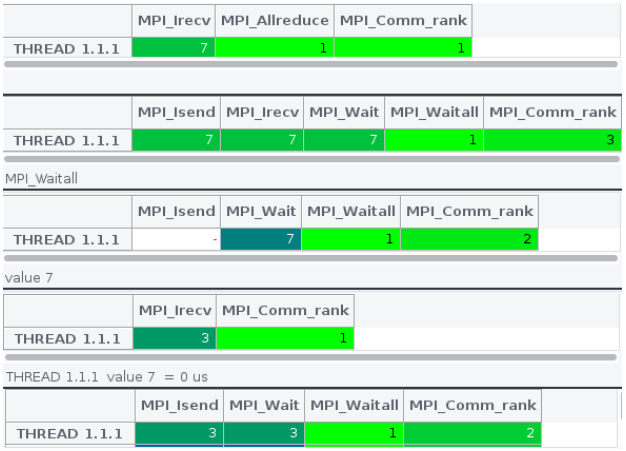


Figure 4.20: Lules2.0 mpi call count for a single iteration splited by computational phases

4.2 Scalability

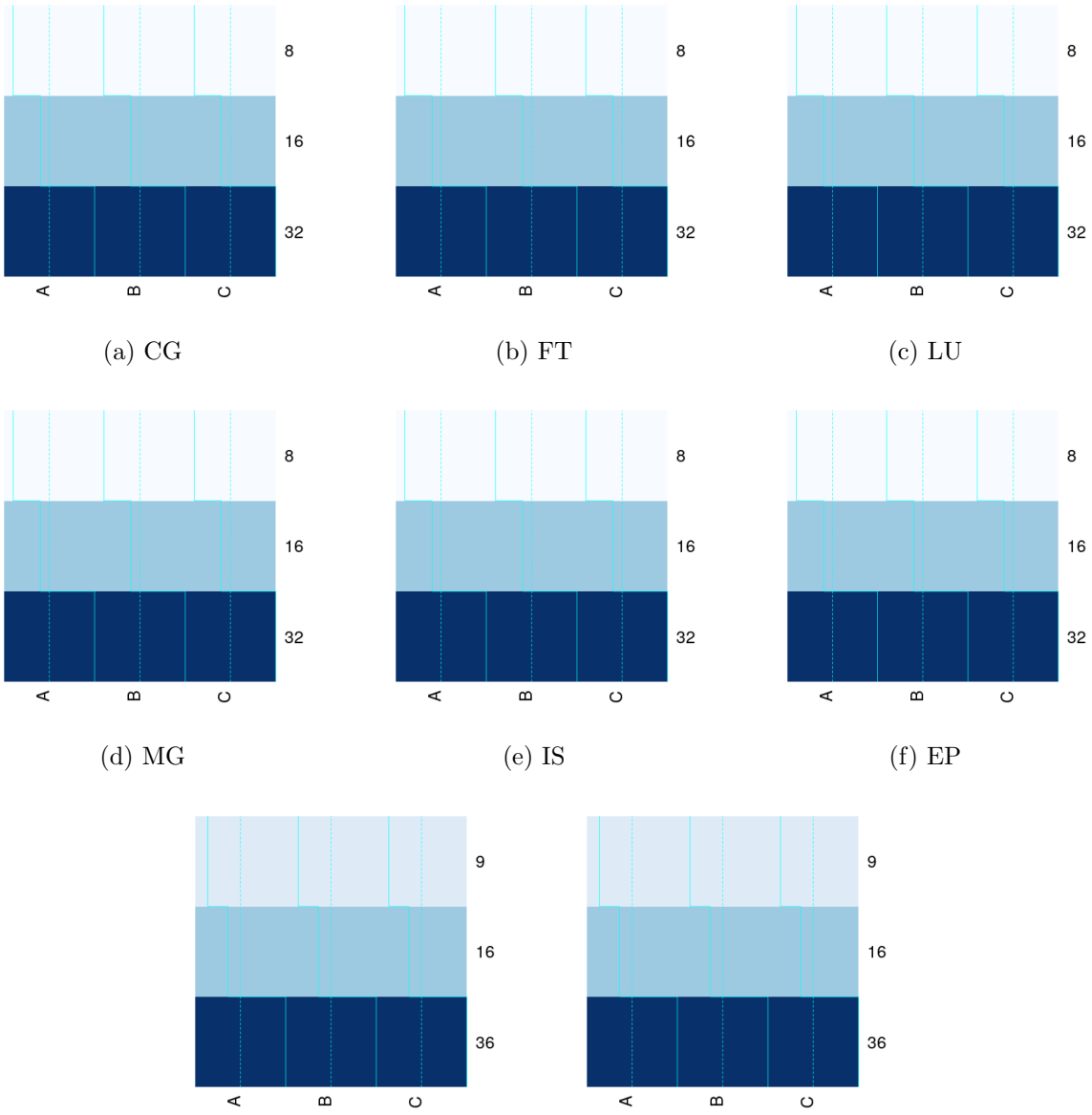
One of the motivations for this work is to be scalable or at least more scalable than the current state of the art. In this section we have driven a little analysis about scalability.

As has been explained previously, the key point of the proposal done in this thesis is the clustering step (section 3.2.2) that is feed by a reduction of the input trace (section 3.2.1). This reduction is the most important point for the scalability of the proposal. What is understood as scalability is that the method does not degrade, or degrade moderately in terms of required resources when the input data grows dramatically. We have argued that since the HPC applications used to be very repetitive, the number of mpi calls to clusterize will remain almost constant despite the size of the execution.

In order to assess our assumptions, a quantitative analysis have been done by executing NPB benchmarks for A,B and C classes and for 8, 16 and 32 ranks (9,16 and 36 for special cases of BT and SP that restrict the number of ranks to squared values, i.e. $1^2, 2^2, 3^2, \cdots$). For every execution first step of the proposed methodology have been executed, i.e. Reduction step and the number of unique mpi calls have been calculated, remember we consider two same call path calls to be different if they have been performed by different processes. This information is plotted in figure 4.21. For every benchmark you can see a matrix where columns are the size of the problem and rows number of processes, the colors indicates the relative quantity of mpi calls, being the darker the greater (for see the numbers go to C.1) .

By a quick peek it can be seen the number of mpi calls remains despite the growing size of the execution time, so the assumption about repetitivity has been demonstrated. On the other axis the trend is quite different since there is an increment on mpi calls number. This phenomena is obvious since we are not merging calls from different processes on first step, so at least they are incrementing with the same factor as the increment of the number of processes.

To check out how the proposed tool is scaling with growing number of processes and growing input, several executions with two different NPB applications have been done. The applications are CG and MG. The interest in these two applications resides in the fact that CG have less unique

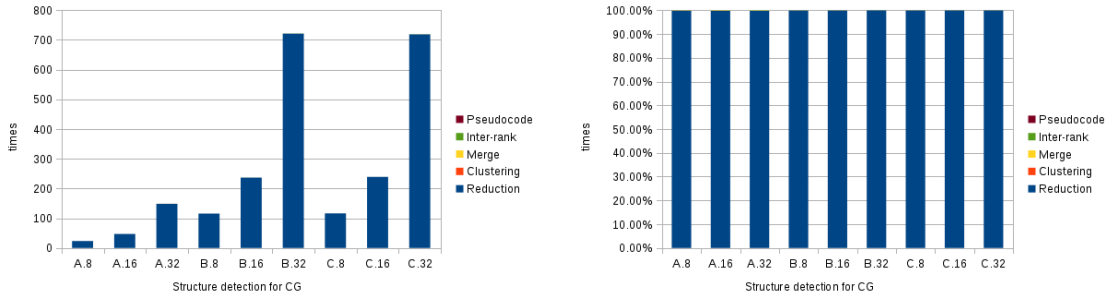


(a) CG (b) FT (c) LU

(d) MG (e) IS (f) EP

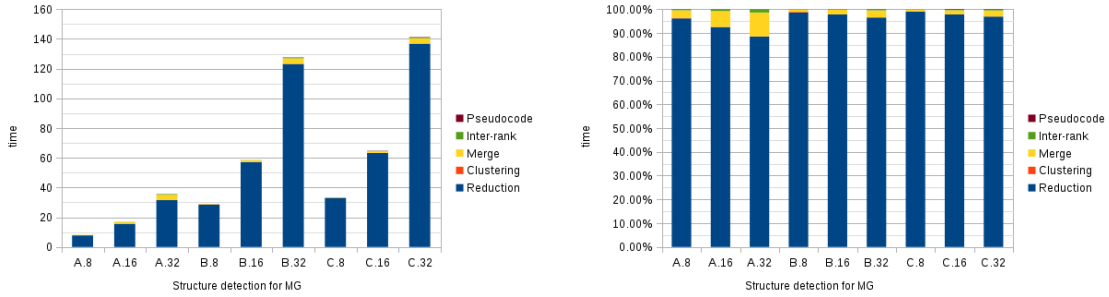
(g) SP (h) BT

Figure 4.21: Number of unique mpi calls



(a) Stacked absolute times (b) Stacked relative times

Figure 4.22: Structure detection times for CG application



(a) Stacked absolute times (b) Stacked relative times

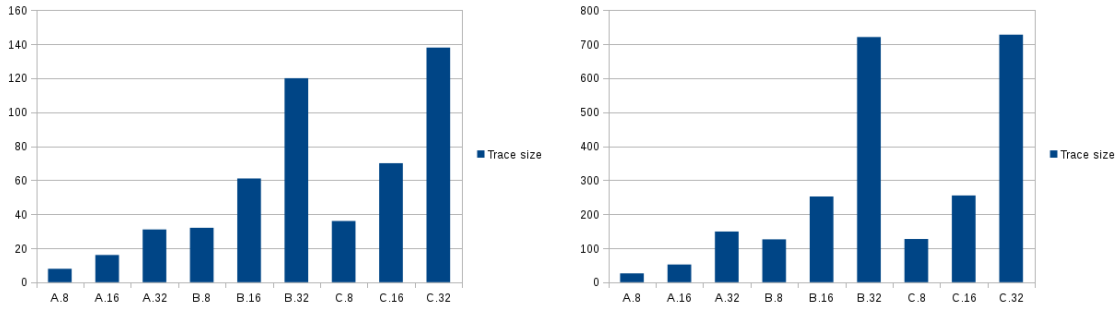
Figure 4.23: Structure detection times for MG application

mpi calls but more repeated and MG have much more unique mpi calls but less repeated. You can see the results on images 4.22 for CG and 4.23 for MG (the numbers can be seen on section C.2.

In both cases can be seen how the overall execution time is highly dominated by the Reduction phase being the responsible for about the 100% of the time for CG (figure 4.22b) and about 90% of the time for the analysis of MG traces (figure 4.23b). About strong scaling, i.e. When the number of processes are being increased but the size of the problem remains the same, it can be seen how time grows linearly with number of ranks in MG (figure 4.23a) but seems to grow faster in case of CG (figure 4.22a). As can be seen on first table in section C.1 the number of mpi calls are just doubling every time scaling on ranks so this is not the reason for this super-linear growth. When scaling the problem size, the execution time of the structure detector grows slowly, it can be said in a logarithmic fashion.

The fundamental reason that explains the Reduction phase times is the size of the input traces as can be seen on 4.24 so as has been introduced in previous chapters, the reduction phase complexity is linear with trace size.

Additionally can be seen how the merge step is much bigger in case of MG than in CG, the reason is that MG has much more nested loops but the manly reason is that since reduction phase is much smaller, the relative weight of the rest of the phases is bigger.



(a) MG Trace sizes

(b) CG Trace size

Figure 4.24: Trace sizes

Chapter 5

Epilog

On the future work and conclusions

5.1 Future work

EVEN if a considerable amount of hours have been invested in this project, as always, there are some parts to be improved to run from this first prototype to a complete and well-polished analysis tool. In this section there are going to explain some aspects that in our opinion should be improved or even extended.

In order to improve the overall performance of an application we have to focus on improving the part of the application that where more time is being wasting on. Trace reduction presents to be the phase that is highly dominating the overall execution time explaining about the 90% of the time. Improve this phase is then a big deal. For sure there should be some improvements to do in the current version, that reads sequentially the whole trace but the real business is on parallelize this process. The sequential read of the trace is not critical for our purposes since every event have a timestamp attached o the order can be reconstructed whenever we want. One possible proposal is to split trace into several files and then parse every one of them independently in parallel, once done the partial results could be merged. Additionally a possible solution could be to work with some big data frameworks like Hadoop what can provide all the reduction infrastructure.

The exposed methodology relies on the mpi calls clustering for extract the applications' structure, so it is a key piece of the overall proposal. The fact of select the best features set to perform this clustering is a difficult business. Our first proposal have been to use number mpi call repetitions and mean time between consecutive mpi calls but as has been explained, in some applications this features leads to aliasing so additional checks have been developed in order to detect and solve them. Having this drawback in mind we have driven an analysis of different features that could improve the behaviour but without positives results. More work have to be done in this sense trying to reduce the cases where aliasing appears by means of analyze in a deeper way new features to use.

About the scaling of clustering phase, also should be explored the alternative to merge same mpi calls from different ranks at the reduce step. It will drive to lighter clusterings since the

number of items will be reduced dramatically for high processes count executions. In principle it have been discarded because we taken a conservative point of view so just in case same mpi call in different processes behave different but since our approach is very focused on SPMD applications can be assumed they will behave similar.

In this proposal only loops with mpi calls in their body can be detected, in order to obtain a more detailed structure, the use of sampling techniques that provides information for these other parts of the executions where no MPI is executed could be explored.

Finally something to explore is to improve the output by developing a graphical user interface that allow to interact with the results in a more natural way.

5.2 Conclusions

In this work have been proposed and developed a new approach for dealing with the extraction of the applications' structure. After revise the literature in this field it has been decided to attack this problem from a different point of view what is indeed the main contribution of this thesis. Here the structure detection problem has been considered as a classification problem instead of a sequential pattern mining one. The main reason to do that is because we are exclusively focused on HPC applications so we decided to exploit their idiosyncrasy that have leads to a sort of ad-hoc methodology for HPC applications that allows to decrease the complexity and so improve the scalability.

To conclude, even if some aspects of the proposal needs to be depurated and more research has to be done in others, in general the results are satisfactory.

5.3 Acknowledgments

At first I would like to be thankful to Prof. Jesús Labarta to bring me the opportunity to do this master thesis at BSC. Specially I want to mention Judit Gimenez, who have provided an important support for the accomplishment of this work. Finally I am thankful to every person from BSC tools team that did not hesitate to aid me during this research.

Appendices

Appendix A

Sequential pattern mining

SEQUENTIAL pattern mining can be defined as: *"Given a set of data sequences, the problem is to discover sub-sequences that are frequent, that is, the percentage of data sequences containing them exceeds a user-specified minimum support."*. Note this definition fits pretty well with our objective of loops recognition in traces so sequential pattern mining is the natural choice. Furthermore there is another definition that fits even better for our problem. It is, referring to patterns to analyze in a temporal sequences: *"... a collection of events that occur relatively close to each other in a given partial order, and ... frequent episodes as a recurrent combination of events"*.

Sequential pattern mining is a technique applied on a wide range of problems like for example predicting systems failure by analyzing a sequence of logs, characterize suspicious behaviour in users by analyzing the sequence of commands entered, for automatically determine "best practices" by analyzing the sequences of actions of an expert, etc so the evolution on this area has been quietly ad-hoc to every problem. On this section pattern mining is introduced and three main classes of algorithms are explained always from an abstract point of view, i.e. without entering into details for specific implementations. This explanations were taken from [Mooney and Roddick, 2013]. Even if the temporal sequences algorithms described in section A.4 seems to be the better choice for structure detection, it has been considered to explain briefly the other types since most of the ideas were first proposed by them.

A.1 Formal notation

Items are literals that belongs to a given alphabet $I = \{i_1, i_2, \ldots, i_m\}$. Then an event is stated as a non-empty unordered set of items $(i_1, i2, \ldots, i_k)$. Finally a sequence is an ordered list of events $\langle \alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_q \rangle$. The ordered metric can be time, space or other. When a sequence is refered as a $k - sequence$ means that this sequence contains k items. A sequence $\langle \alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_n \rangle$ is a subsequence of $\langle \beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \beta_m \rangle$ if there exists integers $i_1, i_2, \ldots i_n$ s.t. $\alpha_1 \subseteq \beta_{i_1}, \alpha_2 \subseteq \beta_{i_2}, \ldots, \alpha_n \subseteq \beta_{i_n}$. So for example $\langle B \rightarrow AC \rangle$ is a subsequence of $\langle AB \rightarrow E \rightarrow ACD \rangle$ being the set of integers 1 and 3.

Having a set of sequences D, *support* or *frequency* of a sequence, denoted as $\sigma(\alpha, D)$, is defined as the number of input sequences in D that contain α. A sequence is *frequent* or not depending on a

threshold named *minimum support*, so is frequent if it happens more than *minimum support* times. The set of frequent k-sequences is denoted as L_k. Moving on, a frequent sequence is *maximal* if it is not subsequence of any other frequent sequence. The task becomes to find all maximal frequent sequences from D.

A.2 Apriori-based algorithms

Mining frequent itemsets is the core of later analysis like mining association rules, correlations, sequential patterns and so on. Apriori first proposal was about discover intra-transaction associations used in database mining, also called knowledge discovery. Being $I = \{i_1, i_2, \ldots, i_m\}$ a set of literals called items and T a transaction $T \subseteq I$ is said T contains X if $X \subseteq T$. Further, an association rule is an implication of the form $X \Rightarrow Y$ where $X \subset I$, $Y \subset I$ and $X \cap Y = \emptyset$. Apriori algorithm was presented in the following paper [Agrawal and Srikant, 1994]. The basis of this algorithm is presented here and several later algorithms were based on this like for example AprioriAll, AprioriSome, DynamicSome, GSP (Generalized Sequential Patterns), PSP and so on. Every one of them are introducing several optimizations and varying mainly the candidates generation step (explained on section A.2.1) but maintains the basic core.

The algorithm consists on two fundamental steps being the first the most challenging one:

1. Find all sets of items (itemsets) that have transaction support above minimum support.

2. Use the large itemsets (itemsets above minimum support) to generate the desired rules.

A.2.1 Discovering large itemsets

The large itemsets discovering implies several passes over the data. The first pass is to find out individual items that are actually large, so which of them appears more than minimum support. On next pass these large items are the seeds, with these seeds the candidate itemsets are generated and the data is passed again in order to find out the large itemsets among the candidates. The same process is repeated until no new large itemsets are found. The basic intuition is that any subset of a large itemset must be large. The algorithm looks like as in pseudocode A.2.1.

The key point in this algorithm is the candidates generation. It is formed by two steps. The first step is to generate all the candidates and the second is to prune those candidates that for sure will not be large itemsets. First step is represented in pseudocode A.2.2 and it can be seen that seed itemsets (from previous pass) are merged in pairs by adding last item from first itemset to the second. Last phase depicted in A.2.3 is about prune the candidates that contain (k-1)-itemsets that do not exists on L_{k-1}. The idea behind that is what has been exposed before, i.e. any subset of a large itemset must be large. This property leads to a powerful pruning. By doing that, the number of candidates is reduced considerably. By this approach is achieved that $C_k \supseteq L_k$. Ideally $C_k = L_k$ so as better the candidates generation is, less verifications (whether the minimum support

is achieved or not) will be done and so better performance.

Algorithm A.2.1: APRIORI ALGORITHM(D)

$L_1 \leftarrow large \quad 1 - itemset$
for $k = 2; L_k \neq \emptyset; k++$
do {
  comment: New candidates
  $C_k \leftarrow aprioriGen(L_{k-1})$
  for all $transactions \quad t \in D$
  do {
    comment: Candidates contained in t
    $C_t \leftarrow subset(C_k, t)$
    for all $candidates \quad c \in C_t$
    do { $c.count++$
  $L_k \leftarrow \{c \in C_k \mid c.count \geq minsup\}$

return $(\bigcup)_k L_k$

Algorithm A.2.2: APRIORI CANDIDATE GENERATOR $1(L_{k-1} - itemsets)$

insert into C_k
select $p.item_1, \ldots, p.item_{k-1}, q.item_{k-1}$
from $L_{k-1} \quad p, L_{k-1} \quad q$
where $p.item_1, \ldots, p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}$
comment: Last condition is for ensuring no duplicates

Algorithm A.2.3: APRIORI CANDIDATE GENERATOR $2(L_{k-1} - itemsets, C_k)$

for all $itemsets \quad c \in C_k$
do {
  for all $(k-1) - subsets \quad s \in c$
  do {
    if $s \notin L_{k-1}$
    then $delete \quad c \quad from \quad C_k$

A.3 Projection-based pattern growth algorithms

Candidates generation presents to be critical for apriori algorithms and even if optimizations in the prune process has been introduced, the generated candidates follows an exponential grow. For example for detect a maximal sequence of 100 elements, $2^{100} \approx 10^{30}$ candidates will be generated. The next problem is that for every step, data needs to be revisited to check out whether new candidates are large itemsets or not.

Pattern growth paradigm presented in [Han et al., 2000] remove completelly the necessity of candidates generation. They achieve improvements on performance for about one order of magnitude respect Apriori-like algorithms explained on previous section by adding two key concepts.

TID	Items Bought	(Ordered) Frequent Items
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

Figure A.1: A transaction database as running example

(i) Frequent pattern tree or FP-tree for short (ii) and FP-tree based pattern mining called FP–growth. Following, this two concepts are explained in more detail.

A.3.1 Frequent pattern tree

The following observations can be used for introduce FP-trees and have been used for its construction. This structure dramatically decrease the size of data to be scanned but maintains all the need information for the mining.

i) One important rule learned from apriori approach is that frequent $(k+1)itemsets$ only can be done from frequent $(k)itemsets$. This observacion leads to the idea of just taking into account frequent $(1)itemsets$ given a minimum support.

ii) These discovered frequent intemsets could be stored in some compact structure, avoiding repeatedly scanning the DB.

iii) Continuing with the idea of compacting important data, it can be said that identical frequent itemsets from different transactions can be merged into one with information about number of occurrences.

iv) And for these partially identical frequent itemsets, shared prefixes can be merged as well.

For improve understandability lets drive a construction of an FP-tree following an example. Imagine we have a database with several transactions like the depicted in figure A.1 (left hand side column). The process ends up with Fp-tree in figure A.2. Lets see how it happens.

First scan of database derives a list of frequent items, i.e. these 1-itemsets above the minimum support value (3 for this example) that is $\langle(f:4),(c:4),(a:3),(b:3),(m:3),(p:3)\rangle$. Note the frequent items in every transaction are on right hand side column in figure A.1. The frequent itemsets here are not sorted by appearance in transaction but by frequence. This sorting will allows more compression on FP-tree construction. The second scan is done over these frequent 1-itemsets and drives the FP-tree construction. First transaction leads to the construction of the first branch (left hand side). Next transaction shares the three first items, so it can be partially merged with first branch. The merge process is just about update the counters and make the new relations. Same process for all transactions. Additionally to the pure tree construction, header table structure is done for ease the task of traverse all possible frequent patterns that contains a given item.

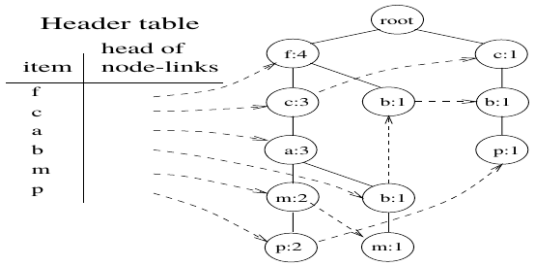


Figure A.2: The FP-tree

A.3.2 Mining FP-trees

FP-growth algorithm is the responsible to find out the frequent patterns by analyzing the FP-tree. The mining starts with 1-itemset analysis. Thanks to the header table all paths for a given item a_i can be get easily. Once all paths, where the given item is involved on, are retrieved a new subtree is build up. Remember that in this process all items below minimum support are pruned. Unlike before now only these retrieved items are taken into account for the counting. This new structure is named a_i conditional pattern base, i.e. the sub-pattern base under the condition of a_i existence. Next step is to call mining function recursively having on every recursive call a large conditional pattern base, so it is growing. It can be better understood by means of an example. Lets follow the previous one.

Starting from the bottom of the header table, lets mine FP-tree for the p item. Two paths arise: $\langle f:4, c:3, a:3, m:2, p:2\rangle$ and $\langle c:1, b:1, p:1\rangle$ (being the number after ":" the occurrences). Note that even if f appears 4 times, only 2 of them appears with p, so the path becomes $\langle f:2, c:2, a:2, m:2, p:2\rangle$. Similarly with second path. Moving on, the construction of the p conditional pattern base is done by counting and pruning these items below minumum support (3 for the example), so the only branch for the new FP-tree is $(c:3)$. Hence only one frequent pattern is derived, i.e. $(cp:3)$. From now to the end, p does not need to be taken into account any more, this is because all possible patterns containing p has been already analyzed. Similarly we can proceed analyzing paths containing m item. Two paths arise: $\langle f:2, c:2, a:2, m:2\rangle$ and $\langle f:1, c:1, a:1, b:1, m:1\rangle$. The new conditional FP-tree just contains the path $\langle f:3, c:3, a:3\rangle$. For show how the pattern is growing, lets see in a deph-first way what recursive calls are done: (i) mine($\langle f:3, c:3, a:3\rangle|m$) (ii) mine($\langle f:3, c:3\rangle|am$) (iii) mine($\langle f:3\rangle|cam$) The frequent pattern derived from this analysis is $(fcam:3)$.

A.4 Temporal sequences

In this section will be shown the basis of these algorithms that concern about the periodicity of a certain patterns over the time (generalizing, over any metric from which the sorting is done). These are obviously the algorithms that best fits to the needs for trace structure detection. First developed framework for datasets considered to be episodic was presented by [Mannila et al., 1995].

Two previous approaches were concerning about the analysis of arbitrary ordered sequences of data, however, this approach considers order as an inherent characteristic of the sequential

structure. This main difference leads to a slightly different approach of sequential pattern mining and introduce new ideas like sliding windows. Nevertheless some important components are shared among them like: (i) Frequency threshold, that is defined as the minimum number of times a sequence have to appear. It is analogous to minimum support of apriori and pattern-growth algorithms. (ii) Relies on generate-and-test paradigm to discover frequent sequences. It is same approach like apriori-like algorithms. (iii) Finally also takes profit from the principle of: all subepisodes are at least as frequent as the superepisode, for candidates generation.

The main objective of these sort of algorithms is: Given a class of episodes, an input sequence of events, a window width, and a frequency threshold, find all episodes of the class that occur frequently enough. Before going to the actual algorithm let's take a look to the main concepts.

A.4.1 Temporal sequences formal notation

On section A.1 have been shown the typical formal notation for pattern mining, now this notation is extended for explaining the new concepts that arise from the temporal sequences minning.

Given a class of elementary event types E_0, an event is a pair (e, t) where $e \in E_0$ and t is an integer that represents the instant when the event appears. An event sequence is a triple $S = (T_s, T^s, S)$ where T_s is the starting time, T^s is the closing time and S is an ordered sequence of events.

A windows on $S = (T_s, T^s, S)$ is a sequence of events $W = (T_w, T^w, W')$ where $T_s \leq T_w$,$T^w \leq T^s$ and W' consists on those events (e_i, t_i) where $T_w \leq t_i < T^w$. The width of windows is $width(W) = T^w - T_w = w$ and the set of all windows in a sequence S is $aw(S, w)$. Episodes are collections of events occurring frequently close to each other, in general, are partially ordered sets of events that can be described as a directed acyclic graph. Are denoted as $\varphi = (V, \leq, g)$ where V is a set of nodes, a partial order $\leq$ on V and a function $g : V \to E_0$ associating each node with an event type. In general V also can contain other episodes forming composite episodes. Episodes can be parallel or sequential. Is parallel when the partial order relation is trivial and an episode is sequential if the partial order relation is a total order. The crucial observation is that all episodes can be described as a composition of parallel and sequential episodes. Last definition is the episode frequency that is described as the ratio between the number of windows containing a given episode and the total number of windows:

$$fr(\varphi, S, w) = \frac{|\{W \in aw(S, w) | \varphi \text{ occurs in } W\}|}{|aw(S, w)|}$$

So an episode is said to be frequent if $fr(\varphi, S, w)$ is above min_freq that is provided by the user.

A.4.2 Algorithm

First step is to find out all frequent episodes in the given sequence, given a class of episodes and a frequency threshold. This part is just like apriori algorithm. The basis of the algorithm presented here is shared with the already presented in section A.2 in the sense that they are based on an iterative process that consists on an alternation between building candidates and recognize frequent episodes by scanning the input data. There is a detail here that makes this phase potentially outperform the naive Apriori-like algorithm. Now we are working with windows, and

we consider just patterns than fits on windows so there is a non-sense to try to get $k - itemsets$ having $k > w$ so the search space is pruned by the windows size. Once all frequent episodes are taken then the second step is about recognizing episodes in sequences. The entire sequence is traversed by a sliding windows and for every one of these windows the analysis looking for episodes is done. Different methods are used for the detection.

i) Parallel episodes: For candidate parallel episode there is a counter that indicates how many events of episode appears into the windows. If the counter is equal to $|\varphi|$ the index of windows is saved because it indicates, the episode has been detected. When the counter is decremented it means that we can add one more windows where this episode is.

ii) Serial episodes: Serial candidate episodes are recognized by using state automata. A new instance of the automata is initialized whenever first event of episode appears on the sliding windows. This automata reach the accepting state when all events are present (and have been arising following a certain order) and is deactivated when the first element that motivates its activation leaves the window. When an automata is removed and there is no other automata for this episode, the number of occurrences is incremented.

Instead of applying a naïve approach where every windows is scanned completely, episodes are recognized in sequences in an incremental fashion. Two adjacent windows are typically very similar so after recognizing episodes in a windows, incremental updates in data structures can be done for the next one.

Like previous algorithms, the exposed above is just he basic idea and more research has drive to better algorithms but maintaining this fundamental idea. Important to mention the Projected Window List presented in [Huang et al., 2004] that it use a sort of pattern-growth fashion for temporal sequences mining for avoid candidate generation.

Appendix B

Automatic code instrumentation

Loops characterization phase of this thesis motivates the development of an automatization of the user source code instrumentation. It is for sure an important piece of this thesis but since it is not the main development have been decided to include it as annex.

As have been previously introduced, this work is done in the Mercurium source-to-source compiler infrastructure by adding a new phase on the source-to-source compilation workflow and it basically consists on inject calls to the Extrae API in order to fire events that i) determine the loops boundaries ii) and additionally fire iteration-level metrics to trace

Before going further, it is the moment to introduce a brief overview about the Mercurium internal structure. In figure B.1 it can be seen the different phases that tooks place when compiling with mercurium. The process starts with one or more than one input source file that are parsed by the parser engine to AST[1] (one per file) that is a commonly used structure for syntax analysis that represents the code in a given file in a hierarchical manner such that every tree level corresponds to a nested level in code. Every one of these ASTs are then modified by the different phases on the Compiler phase pipeline. After all AST modifications are done, it is checked for correctness and parsed to code again (prettyprinting) that leads to the "Output Source". Finnaly output file/s are compiled as usual by the backend compiled (such as GCC, ICC, ...) and the executable file is ready to be executed.

The developments presented in this chapter are done exclusively on the "Compiler phases pipeline" by adding a new phase. The infrastructure provides an input and expects an output, both are AST so the work to do is to traverse the AST looking for those target parts (target pieces of code) to transform, made modifications on them and return it as the output. Since we are concerned about loops we just need to look for loops (for, while, do while,...), for this purpose Mercurium infrastructure provides a useful mechanism that traverse the tree and gives you a way to program callbacks that will be called every time the structure you are aware on is encountered. These callbacks are programmed in such a way that inject monitors to loops For simplify the transformations, these calls are not done directly to the trace library but to an intermediate helper library that will do all stuff and just expose simple calls.

On next sections the work done for the two developments are deeply explained.

[1] Abstract Syntax Tree

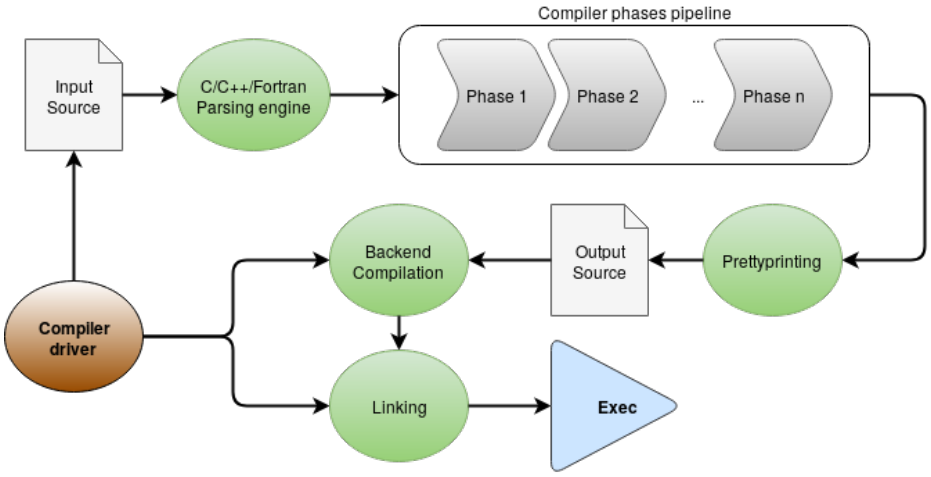


Figure B.1: Mercurium internals overview

B.1 Instrumentation for PCA analysis

For PCA analysis what is needed is gather information from loops and its iterations so the transformations done looks like the transformation of pseudo-code B.1.1 to B.1.2.

Algorithm B.1.1: ORIGINAL LOOP()

for $i \in I$
do $\{ SomeWork()$

Algorithm B.1.2: TRANSFORMED LOOP()

$MLoopInit(loop_{line}, loop_{file})$
for $i \in I$
do $\begin{cases} MIterInit(chance) \\ SomeWork() \\ MIterFini() \end{cases}$
$MLoopFini(loop_{line}, loop_{file})$

It can be seen that there are entry and exit calls both on loops and iterations. In case of loops the arguments are the needed for identify the loop unambiguously. In case of iterations loop identifier is not needed because since trace holds temporal information is quite easy to determine to what loop every iteration belongs, instead of it the argument is the chance[2] to a given iteration, to be instrumented or not. The decision of instrument an iteration given a probability arise from the fact that extract information of all loops at level of iterations adds an unmanageable overhead to the trace size and the executions used to be very repetitive so taking just few iterations should be enough for figure out the global behaviour of all iterations. Summarizing, the main ideas for the instrumentation are:

i) For every loop entry and exit are marked in trace.

ii) Additionally, before the exit event, the total number of iterations performed is fired to trace.

[2]This chance can be set statically at compilation time and dynamically at execution time by means of an environment variable.

iii) An iteration will be instrumented or not depending on a given probability.

iv) For every instrumented iteration entry and exit are marked in trace with additional information such as some hardware counters.

v) Nested loops will be instrumented depending on the decission taken for the parent iteration. Take into account that this fact implies that the probability for a nested loop iteration to be instrumented is not *chance* but $chance^{nestedLevel}$ with $chance \in [0, 1)$.

In pseudo-codes B.1.3 and B.1.4 it can be seen how the calls that marks loops boundaries works and on pseudo-codes B.1.5 and B.1.6 calls that determines the iterations boundaries can be found.

Algorithm B.1.3: MLoopInit($file, line$)

$instrumentLoop \leftarrow True$
if $size(decissionStack) > 0$
then $\begin{cases} ds \leftarrow top(decissionStack) \\ instrumentLoop \leftarrow ds \end{cases}$
if $instrumentLoop$
then $\begin{cases} hash \leftarrow hash(loop_{file}, loop_{line}) \\ ExtraeEvent(LOOPINIT, hash) \\ push(iterCounterStack, 0) \end{cases}$

Algorithm B.1.4: MLoopFini($file, line$)

$instrumentLoop \leftarrow True$
if $size(decissionStack) > 0$
then $\{instrumentLoop \leftarrow top(decissionStack)$
if $instrumentLoop$
then $\begin{cases} ic \leftarrow pop(iterCounterStack) \\ hash \leftarrow hash(loop_{file}, loop_{line}) \\ ExtraeEvent(LOOPITERS, ic) \\ ExtraeEvent(LOOPFINI, hash) \end{cases}$

Algorithm B.1.5: MIterInit($chance$)

$instrumentIter \leftarrow True$
$topInstrumentIter \leftarrow True$
$r \in U(0, 1)$
if $size(decissionStack) > 0$
then $\begin{cases} ds \leftarrow top(decissionStack) \\ topInstrumentIter \leftarrow ds \end{cases}$
if $topInstrumentIter$
then $\begin{cases} instrumentIter \leftarrow (r < chance) \\ top(iterCounterStack) + + \\ \textbf{if } instrumentIter \\ \quad \textbf{then } \begin{cases} ic \leftarrow top(iterCounterStack) \\ ExtraeEventHWC(IINIT, ic) \end{cases} \end{cases}$
$d \leftarrow instrumentIter \& topInstrumentIter$
$push(decissionStack, d)$

Algorithm B.1.6: MIterFini()

$instrumentIter \leftarrow pop(decissionStack)$
if $size(decissionStack) > 0$
then $\{ExtraeEventAndCounters(IFINI)$

The nesting level of a given loop can not be determined statically since every file have its own AST so it have to be managed dynamically. In order to do that we rely on the use of stacks that seems to be the natural choice, there are two stacks needed here: i) decissionStack that holds the

information whether one iteration is instrumented or not ii) and iterCounterStack that holds the number of total iterations. Former is used to decide if a given iteration and its subloops (and its iterations) should be instrumented or not, while the last is used to fire the number of actual iterations performed whether have been instrumented or not.

B.2 Instrumentation for Variable Importance analysis

Variable importance analysis needs every mpi call to be labeled with the identificator of the loop where every one of them lies. To do so the followed strategy have been instrument every entry and exit of every loop, by injecting calls to the helper library that will keep track about what loop or nested loop the execution is traversing. By this way, on any instant the information about where the execution is in terms of loops or nested loops is available. Then just before every mpi call there is a call that fires this information to the trace. This information will be later post-process such that every event containing loop ids are providing information for the next mpi call.

Appendix C

Experiments

In this appendix it can be checked out the numbers used for the plots showed up on the results chapter.

C.1 Number of unique mpi calls

Table C.1: Number of unique mpi calls

	A.8	A.16	A.32	B.8	B.16	B.32	C.8	C.16	C.32
BT	945	1680	3780	945	1680	3780	945	1680	3780
CG	480	960	1920	480	960	1920	480	960	1920
EP	72	144	288	72	144	288	72	144	288
FT	121	241	481	121	241	481	121	241	481
LU	635	1421	2993	635	1421	2993	635	1421	2993
IS	117	237	477	117	237	477	117	237	477
MG	1794	3344	6656	1794	3344	6656	1794	3344	6656
SP	819	1456	3276	819	1456	3276	819	1456	3276

C.2 Structure detection execution times

Table C.2: Structure detection times for CG

	A.8	A.16	A.32	B.8	B.16	B.32	C.8	C.16	C.32
Reduction	23.25	47.08	148.3	115.55	236.8	721.25	116.42	239.1	719.01
Clustering	0.002	0.008	0.01	0.002	0.025	0.01	0.002	0.02	0.01
Merge	0.03	0.08	0.22	0.1	0.179	0.4	0.08	0.15	0.47
Inter-rank	0.007	0.02	0.12	0.007	0.02	0.12	0.007	0.029	0.11
Pseudocode	0.006	0.007	0.01	0.009	0.007	0.002	0.006	0.007	0.01

Table C.3: Structure detection times fro MG

	A.8	A.16	A.32	B.8	B.16	B.32	C.8	C.16	C.32
Reduction	7.8	15.6	31.5	28.5	57.13	123.15	33.04	63.28	136.8
Clustering	0.007	0.014	0.04	0.08	0.01	0.05	0.008	0.02	0.04
Merge	0.27	1.13	3.57	0.29	1.15	3.8	0.29	1.13	3.6
Inter-rank	0.03	0.12	0.49	0.03	0.12	0.48	0.03	0.12	0.5
Pseudocode	0.008	0.01	0.01	0.01	0.015	0.14	0.008	0.16	0.25

Bibliography

[Agrawal and Srikant, 1994] Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules.

[Agrawal and Srikant, 1995] Agrawal, R. and Srikant, R. (1995). Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14. IEEE Comput. Soc. Press.

[Aguilar et al., 2014] Aguilar, X., Fürlinger, K., and Laure, E. (2014). Mpi trace compression using event flow graphs. In *European Conference on Parallel Processing*, pages 1–12. Springer.

[Aguilar et al., 2016] Aguilar, X., Fürlinger, K., and Laure, E. (2016). Event flow graphs for mpi performance monitoring and analysis. In *Tools for High Performance Computing 2015*, pages 103–115. Springer.

[Casas et al., 2008] Casas, M., Badia, R., and Labarta, J. (2008). Automatic analysis of speedup of mpi applications. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 349–358, New York, NY, USA. ACM.

[Casas et al., 2007] Casas, M., Badia, R. M., and Labarta, J. (2007). Automatic structure extraction from mpi applications tracefiles. In *European Conference on Parallel Processing*, pages 3–12. Springer.

[González García, 2013] González García, J. (2013). Application of clustering analysis and sequence analysis on the performance analysis of parallel applications.

[Han et al., 2000] Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1–12. ACM.

[Huang et al., 2004] Huang, K.-Y., Chang, C.-H., and Lin, K.-Z. (2004). Prowl: An efficient frequent continuity mining algorithm on event sequences. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 351–360. Springer.

[Jouppi et al., 2017] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis

of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM.

[Knupfer and Nagel, 2005] Knupfer, A. and Nagel, W. E. (2005). Construction and compression of complete call graphs for post-mortem program trace analysis. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 165–172. IEEE.

[Llort Sánchez, 2015] Llort Sánchez, G. (2015). Intelligent instrumentation techniques to improve the traces information-volume ratio.

[López-Cueva et al., 2012] López-Cueva, P., Bertaux, A., Termier, A., Méhaut, J. F., and Santana, M. (2012). Periodic pattern mining of embedded multimedia application traces. In *Lecture Notes in Electrical Engineering*, volume 181 LNEE, pages 29–37.

[Malony et al., 2005] Malony, A. D., Shende, S., and Morris, A. (2005). Phase-based parallel performance profiling. In *PARCO*, pages 203–210.

[Mannila et al., 1995] Mannila, H., Toivonen, H., and Verkamo, A. I. (1995). Discovering frequent episodes in sequences extended abstract. In *1st Conference on Knowledge Discovery and Data Mining*.

[Mooney and Roddick, 2013] Mooney, C. H. and Roddick, J. F. (2013). Sequential pattern mining–approaches and algorithms. *ACM Computing Surveys (CSUR)*, 45(2):19.

[Moore, 1965] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[Noeth et al., 2009] Noeth, M., Ratn, P., Mueller, F., Schulz, M., and de Supinski, B. R. (2009). Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710.

[Patt, 2017] Patt, Y. N. (2017). Computer architecture principles and tradeoffs. Seminar lecture.

[Patt et al., 1985] Patt, Y. N., Hwu, W. M., and Shebanow, M. (1985). Hps, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, MICRO 18, pages 103–108, New York, NY, USA. ACM.

[Rokach and Maimon, 2005] Rokach, L. and Maimon, O. (2005). Clustering methods. In *Data mining and knowledge discovery handbook*, pages 321–352. Springer.

[Safyallah and Sartipi, 2006] Safyallah, H. and Sartipi, K. (2006). Dynamic Analysis of Software Systems using Execution Pattern Mining. *The 14th IEEE International Conference on Program Comprehension (ICPC '06)*, pages 84–88.

[Saviankou et al., 2015] Saviankou, P., Knobloch, M., Visser, A., and Mohr, B. (2015). Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352.

[Söderlind, 2017] Söderlind, S. (2017). Application performance evaluation using deep learning. Master's thesis, Barcelona Schools of Informatics, UPC, Barcelona, Spain.

[Top500, 2017] Top500 (2017). The list (june 2017). https://www.top500.org/list/2017/06/. Accessed: 2017-09-22.

[Trahay et al., 2015] Trahay, F., Brunet, E., Bouksiaa, M. M., and Liao, J. (2015). Selecting points of interest in traces using patterns of events. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 70–77. IEEE.

[Wolf and Mohr, 2003] Wolf, F. and Mohr, B. (2003). Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture*, 49(10):421–439.

[Zhao et al., 2008] Zhao, C., Ates, K., Kong, J., and Zhang, K. (2008). Discovering program's behavioral patterns by inferring graph-grammars from execution traces. In *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, volume 2, pages 395–402.